A Practical Mode System for Recursive Definitions

ALBAN REYNAUD, ENS Lyon, France
GABRIEL SCHERER, INRIA, France
JEREMY YALLOP, University of Cambridge, United Kingdom

In call-by-value languages, some mutually-recursive definitions can be safely evaluated to build recursive functions or cyclic data structures, but some definitions (`let rec x = x + 1`) contain vicious circles and their evaluation fails at runtime. We propose a new static analysis to check the absence of such runtime failures.

We present a set of declarative inference rules, prove its soundness with respect to the reference source-level semantics of Nordlander, Carlsson, and Gill [2008], and show that it can be directed into an algorithmic backwards analysis check in a surprisingly simple way.

Our implementation of this new check replaced the existing check used by the OCaml programming language, a fragile syntactic criterion which let several subtle bugs slip through as the language kept evolving. We document some issues that arise when advanced features of a real-world functional language (exceptions in first-class modules, GADTs, etc.) interact with safety checking for recursive definitions.

CCS Concepts: • **Software and its engineering → General programming languages**; **Recursion**.

Additional Key Words and Phrases: recursion, call-by-value, types, semantics, ML, functional programming

ACM Reference Format:
Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. 2021. A Practical Mode System for Recursive Definitions. *Proc. ACM Program. Lang.* 5, POPL, Article 45 (January 2021), 29 pages. Author version with appendices.

1 INTRODUCTION

Recursion pervades functional programs. Functional programmers often start out by writing simple recursive definitions such as the Fibonacci function, shown here in OCaml:

```
let rec fib = fun x -> if x <= 1 then x
                       else fib (x-1) + fib (x-2)
```

This definition is elegant but, alas, impractical: computing `fib n` takes time exponential in `n`. One way to improve performance is to memoize: in place of a function, we might (recursively) define a lazy list, `lfibs`, whose *n*th element represents `fib n`:

```
let rec lfibs = lazy (0 :: lazy (1 :: map2 (+) lfibs (tail lfibs)))
```

As these definitions show, recursion is useful for defining both functions and values. However, lazy-list memoization is rarely used in eager languages, since elegance suffers from the need to make all laziness explicit. Here is a more idiomatic memoized function `mfib`, mutually-defined with a record `mfibs`, which pairs the function with a memo-table of its previously-computed values:

Authors' addresses: Alban Reynaud, ENS Lyon, France; Gabriel Scherer, INRIA, France; Jeremy Yallop, University of Cambridge, United Kingdom.

2475-1421/2021/1-ART45

```
let rec mfib = fun x -> if x <= 1 then x
                        else remember mfibs (x-1) + remember mfibs (x-2)
    and mfibs = { f = mfib; values = empty_table () }
```

(The `remember` function retrieves previously-computed values and computes and stores new entries.) This definition exposes both `mfib` and its table `mfibs`, risking inadvertent modification of the table. A cautious programmer might avoid this danger by making `mfibs` local to `mfib`:

```
let rec mfib' = let mfibs' = { f = mfib'; values = empty_table () } in
                fun x -> if x <= 1 then x
                         else remember mfibs' (x-1) + remember mfibs' (x-2)
```

1.1 Recursion: Expressiveness vs Safety

As this tapestry of `fibs` suggests, the *usefulness* of recursion is not limited to simple function definitions: our examples build record values, call functions and bind local names. However, in eager languages not all recursion is *safe*. For example, here is an unsafe eager variant of `lfibs`:

```
let rec efibs = 0 :: 1 :: map2 (+) efibs (tail efibs) (* unsafe! *)
```

Evaluation of `efibs` fails, since `map2` and `tail` access `efibs` before it is fully constructed.

Existing functional languages incorporate various approaches to balancing usefulness and safety.

In some languages, such as Scheme [Sperber et al. 2009], mutually-recursive bindings are evaluated immediately, and it is a run-time error for evaluation to encounter any identifier being bound. Evaluating `mfibs`, which refers to `mfib`, would produce such an error.

Other languages, such as F♯, provide a kind of lazy evaluation for recursive value bindings [Syme 2006] (discussed in §8) which supports the construction of recursive objects. F♯'s approach allows more programs to execute without error — it is sufficient to support `mfib`, though not `mfib'` — but it does not entirely eliminate run-time errors from ill-formed recursive bindings. Consequently, an additional syntactic check [Syme 2005] rejects some cases of self-reference that would result in run-time errors. (However, even this additional check is not sufficient to eliminate *all* such errors.)

Finally, some languages, such as Standard ML [Milner et al. 1997], incorporate a more severe approach, permitting recursion only through syntactic function definitions such as `fib`, and statically rejecting `mfib`, `mfib'` and `efib`. (Some implementations also support laziness, and allow `lfibs`.)

Of these design choices, Standard ML's most fully embodies Milner's dictum: *well-typed programs do not go wrong*. However, as we show in this paper, Standard ML's treatment of recursive definitions is unnecessarily restrictive: it is possible to define a much more permissive criterion for recursive definitions that still ensures the absence of run-time errors. Our criterion allows useful definitions such as `lfibs`, `mfib` and `mfib'`, while rejecting incorrect programs such as `efibs`. We present the criterion as a mode system (proved sound with respect to an operational semantics), suitable for incorporating into a compiler — indeed, our implementation has already been merged into the mainline OCaml compiler.

1.2 OCaml Needed Fixing

Before the work described in this paper, OCaml took an approach similar to F♯'s (although somewhat more precise, and based on OCaml's existing eager semantics rather than a translation into thunks), checking for vicious recursive definitions via syntactic analysis of an intermediate representation of programs. We believe OCaml's check as originally defined[1] was correct, but it proved fragile and difficult to maintain as the language evolved and new features interacted with recursive definitions. Over the years, several bugs were found where the check was unduly lenient (§2.3). In conjunction

[1] https://ocaml.org/releases/4.05/htmlman/extn.html#sec217

with OCaml's efficient compilation scheme for recursive definitions [Hirschowitz et al. 2009], this leniency resulted in memory safety violations, and led to segmentation faults when definitions that accessed recursively-defined objects before they were initialized were allowed through undetected.

1.3 Generalized Recursion in Practice

In order to determine whether generalized recursive definitions were used in practice, we searched a large subset of packages in the OCaml software repository (OPAM) for instances of generalized recursion (i.e. recursive definitions that Standard ML would reject). At the time of our analysis in July 2020, we found 309 distinct examples of such definitions in around 74 packages. The definitions variously made use of local bindings and other local constructs (such as local exception definitions), evaluation of sub-terms, record and variant construction and laziness, in both singly- and mutually-recursive bindings. Generalized recursion definitions are used in useful data structures and important libraries that many other packages in turn depend on. We found that 1613 packages out of 2819 in the repository at the time (57%) depend on at least one of those packages using generalized recursion.

1.4 Our Analysis

The present document formally describes our analysis using a core ML language (§3). We present inference rules (§4) and study the meta-theory of the analysis. We propose a source-level operational semantics, refreshing semantics proposed in earlier works [Ariola and Felleisen 1997; Hirschowitz, Leroy, and Wells 2009; Nordlander, Carlsson, and Gill 2008] with explicit substitutions using *reduction at a distance* (§5), and show that our analysis is sound for this semantics. We also propose a semantics that uses mutable updates to a global store, closer to production-compiler compilation strategies (§6), for which our analysis is also sound. Finally, we discuss the challenges caused by scaling the analysis to OCaml (§7), a full-fledged functional language, in particular the delicate interactions with non-uniform value representations (§7.2), with exceptions and first-class modules (§7.3), and with Generalized Algebraic Datatypes (GADTs) (§7.4).

1.5 Adoption in OCaml

The initial version of our new system was originally released in OCaml 4.06.0 (3 Nov 2017). We formalised the new system after its release, and reworked the implementation to better match the formalisation. The updated implementation was released in OCaml 4.08.0 (13 June 2019).

Before releasing our new system, we sought to determine whether it was significantly more restrictive in practice than the previous check by running it over the packages in OPAM. This investigation did not uncover any code that was accepted by the previous check but rejected by our new system. Six major OCaml releases later there has been only one report of such a program (#7767); it was straightforward to update our system to accept it.

Of course, since we looked only at existing OCaml code, our investigation could not uncover definitions that were rejected by the previous check but accepted by our new system, which is more permissive in some cases (§2.3). However, our goal was not to increase the expressivity of OCaml's value definitions, but to design a static check that was backwards-compatible with the previous check, while being easier to reason about and evolve in tandem with the language (§7). Our aim was to make the check as simple as possible within the constraints, not as expressive as possible.

Furthermore, moving the check from the compiler middle end into the type checker has another benefit: it is convenient for tools that reuse OCaml's type-checker without performing compilation, such as the multi-stage language MetaOCaml [Kiselyov 2014] (which type-checks code quotations) and the Merlin language server [Bour et al. 2018] (which type-checks code during editing).

1.6 Contributions

We claim the following contributions:

- We propose a new system of inference rules that captures the safety conditions for recursive definitions in an eager language (§4), previously enforced in OCaml by ad-hoc syntactic restrictions.
- We prove the analysis sound with respect to a source-level operational semantics: accepted recursive terms evaluate without vicious-circle failures (§5). Our source-level semantics is justified by a simulation result with lower-level backpatching semantics with a global store (§6), for which our analysis is also sound.
- We have implemented a checker derived from the rules, scaled up to the full OCaml language (§7) and integrated in the OCaml implementation.
- Our analysis is less fine-grained on functions than existing works (§8), thanks to a less demanding problem domain (ML functions rather than ML functors), but in exchange it provides finer-grained handling of cyclic data and an effective inference algorithm.

2 OVERVIEW

2.1 Access Modes

Our analysis is based on the classification of each use of a recursively-defined variable using "access modes" or "usage modes" m. These modes represent the degree of access needed to the value bound to the variable during evaluation of the recursive definition.

For example, in the recursive function definition

```
let rec f = fun x -> ... f ...
```

the recursive reference to `f` in the right-hand-side does not need to be evaluated to define the function value `fun x ->`... since its value will only be required later, when the function is applied. We say that, in this right-hand-side, the mode of use of the variable `f` is Delay.

In contrast, in the vicious definition `let rec x = 1 + x` evaluation of the right-hand side involves accessing the value of `x`; we call this usage mode a Dereference. Our static check rejects mutually-recursive definitions that access recursively-bound names under this mode.

Some patterns of access fall between the extremes of Delay and Dereference. For example, in the cyclic datatype construction `let rec obj = { self = obj }` the recursively-bound variable `obj` appears on the right-hand side without being placed inside a function abstraction. However, since it appears in a "guarded" position, within the record constructor `{self = -}`, evaluation only needs to access its address, not its value. We say that the mode of use of the variable `ones` is Guard.

Finally, a variable `x` may also appear in a position where its value is not inspected, neither is it guarded beneath a constructor, as in the expression `x`, or `let y = x in y`, for example. In such cases we say that the value is "returned" directly and use the mode Return. As with `Dereference`, recursive definitions that access variables at the mode Return, such as `let rec x = x`, would be under-determined and are rejected.

We also use a last Ignore mode to classify variables that are not used at all in a term.

2.2 An Inference System (and Corresponding Backwards Analysis)

The central contribution of our work is a simple system of inference rules for a judgment of the form $\Gamma \vdash t : m$, where t is a program term, m is an access mode, and the environment Γ maps term variables to access modes. Modes classify terms and variables, playing the role of types in usual type systems. The example judgment $x : \text{Dereference}, y : \text{Delay} \vdash (x + 1, \text{lazy } y) : \text{Guard}$ can be read alternatively

forwards: If we know that x can safely be used in Dereference mode, and y can safely be used in Delay mode, then the pair $(x + 1, \text{lazy } y)$ can safely be used under a value constructor (in a Guard-ed context).

backwards: If a context accesses the program fragment $(x + 1, \text{lazy } y)$ under the mode Guard, then this means that the variable x is accessed at the mode Dereference, and the variable y at the mode Delay.

This judgment uses access modes for two purposes: to classify variables, and to classify the constraints imposed on a subterm by its surrounding context. If a context $C[\Box]$ uses its hole $\Box$ at the mode m, then any derivation for the plugged context $C[t] : \text{Return}$ will contain a sub-derivation of the form $t : m$ for the term t.

In general, we can define a notion of mode composition: if we try to prove $C[t] : m'$, then the sub-derivation will check $t : m'\,[m]$, where $m'\,[m]$ is the composition of the access-mode m under a surrounding usage mode m', and Return is neutral for composition.

Our judgment $\Gamma \vdash t : m$ can be directed into an algorithm following our backwards interpretation. Given a term t and a mode m as inputs, our algorithm computes the least demanding environment Γ such that $\Gamma \vdash t : m$ holds.

For example, the inference rule for function abstractions in our system is as follows:

$$\frac{\Gamma, x : m_x \vdash t : m\,[\text{Delay}]}{\Gamma \vdash \lambda x.\, t : m}$$

The backwards reading of the rule is as follows. To compute the constraints Γ on $\lambda x.\, t$ in a context of mode m, it suffices to check the function body t under the weaker mode $m\,[\text{Delay}]$, and remove the function variable x from the collected constraints — its mode does not matter. If t is a variable y and m is Return, we get the environment $y : \text{Delay}$ as a result.

Given a family of mutually-recursive definitions $\texttt{let rec } (x_i = t_i)^{i \in I}$, we run our algorithm on each t_i at the mode Return, and obtain a family of environments $(\Gamma_i)^{i \in I}$ such that all the judgments $(\Gamma_i \vdash t_i : \text{Return})^{i \in I}$ hold. The definitions are rejected if one of the Γ_i contains one of the mutually-defined names x_j under the mode Dereference or Return rather than Guard or Delay.

2.3 Issues with the Previous Check

Before this work, the safety criterion used by OCaml for recursive value definitions was an ad-hoc grammatical restriction, formulated essentially as a context-free grammar of accepted definitions (see its description in the reference manual). Furthermore, this syntactic check was not performed on the source program directly, but on an intermediate representation (the Lambda code) — so that it wouldn't have to take into account various surface-language forms that desugar to the same intermediate-language construct.

We list below some of the known issues with the previous check. They were solved by our work.

#7231: unsoundness with nested recursive bindings. The previous check accepted the following unsafe program.

```
let rec r = let rec x () = r
                and y () = x ()
            in y ()
      in r "oops" (* segfault *)
```

The problem is that while the declarations of `x` and `y` are "safe" (in some sense) with respect to `r`, using `y` is not safe — it returns `r` itself. This subtlety was lost on the previous check. With the current check, `y ()` uses `r` at mode `Return`, which is stricter than Guard, so this program is rejected.

#7215: unsoundness with GADTs. The previous check accepted the following unsafe program.

```
let is_int (type a) =
  let rec (p : (int, a) eq) = match p with Refl -> Refl in p
```

This program uses a recursive value declaration of a GADT value to build a type-equality between `int` and an arbitrary type `a`. Our check rejects the program because `match p with Refl -> ...` is a dereferencing use of `p`. The previous check was run on an intermediate form, after various optimizations, one of which would eliminate the single-case match away, resulting in the (unsound) program passing the check.

#6939: unsoundness with float arrays.

```
let rec x = ([| x |]; 1.) in ()
```

This program defines `x` to be the floating-point value `1.` after ignoring the value of the one-element array `[| x |]`. Although the program was accepted by the previous check, OCaml's non-uniform value representation makes it unsafe, and it would fail with a segmentation fault when run, as explained in Section 7.2 (Dynamic Representation Checks: Float Arrays). Our algorithm uses typing information, which is needed to detect this case: construction of a float array is treated as a Dereference context for its elements.

#4989: inconveniently rejected program.

```
let rec f = let g = fun x -> f x in g
```

This program, which gives a local name to an expression that accesses `f` at mode Delay, is perfectly safe, but was rejected by the previous check. A grammar-based check lacks a form of composability that would allow the use of local bindings to give names to sub-expressions in an analysis-preserving way. This issue (#4989) dates back to 2010: this form of composability had been requested by users for a long time as a convenience feature, but the previous check could not be extended to allow it. On the contrary, proper handling of inner **`let`**-bindings falls out naturally from our type-system-inspired approach.

3 A CORE LANGUAGE OF RECURSIVE DEFINITIONS

Family notation. We write $(\ldots)^{i \in I}$ for a family of objects parametrized by an index i over finite set I, and $\emptyset$ for the empty family. Furthermore, we assume that index sets are totally ordered, so that the elements of the family are traversed in a predetermined linear order; we write $\left(t_{i_1}\right)^{i_1 \in I_1}, \left(t_{i_2}\right)^{i_2 \in I_2}$ for the combined family over $I_1 \uplus I_2$, with the indices in I_1 ordered before the indices of I_2. We often omit the index set, writing $(\ldots)^i$. Families may range over two indices (the domain is the cartesian product), for example $\left(t_{i,j}\right)^{i,j}$.

Our syntax, judgments, and inference rules will often use families: for example, $\texttt{let rec}\ (x_i = t_i)^i$ is a mutually-recursive definition of families $(t_i)^i$ of terms bound to corresponding variables $(x_i)^i$ — assumed distinct, following the Barendregt convention. Sometimes a family is used where a term is expected, and the interpretation should be clear: when we say "$(\Gamma_i \vdash t_i : m_i)^i$ holds", we implicitly use a conjunctive interpretation: each of the judgments in the family holds.

3.1 Syntax

Figure 1 introduces a minimal subset of ML containing the interesting ingredients of OCaml's recursive values:

- A multi-ary **`let rec`** binding $\texttt{let rec}\ (x_i = t_i)^i\ \texttt{in}\ u$.
- Functions (λ-abstractions) $\lambda x.\, t$ to write recursive occurrences whose evaluation is delayed.

$$
\begin{array}{lll}
\text{Terms} \ni t, u ::= & x, y, z & \\
& \mid \texttt{let rec}\ b\ \texttt{in}\ u & \text{Bindings} \ni b ::= (x_i = t_i)^i \\
& \mid \lambda x.\, t \mid t\ u & \text{Handlers} \ni h ::= (p_i \to t_i)^i \\
& \mid K\,(t_i)^i \mid \texttt{match}\ t\ \texttt{with}\ h & \text{Patterns} \ni p, q ::= K\,(x_i)^i
\end{array}
$$

Fig. 1. Core language syntax

- Datatype constructors $K\,(t_1, t_2, \dots)$ to write (safe) cyclic data structures; these stand in both for user-defined constructors and for built-in types such as lists and tuples.
- Shallow pattern-matching ($\texttt{match}\ t\ \texttt{with}\ \left(K_i\,(x_{i,j})^j \to u_i\right)^i$), to write code that inspects values, in particular code with vicious circles.

The following common ML constructs do not need to be primitive forms, as we can desugar them into our core language. In particular, the full inference rules for OCaml (and our check) exactly correspond to the rules (and check) derived from this desugaring.

Besides dispensing with many constructs whose essence is captured by our minimal set, we further simplify matters by using an untyped ML fragment: we do not need to talk about ML types to express our check, or to assume that the terms we are working with are well-typed.[2] However, we do assume that our terms are well-scoped — note that, in $\texttt{let rec}\ (x_i = v_i)^i\ \texttt{in}\ u$, the $(x_i)^i$ are in scope of u but also of all the v_i.

Remark 1. *Recursive values are a controversial feature as they break the assumption that structurally-decreasing recursive functions will terminate on all inputs. The uses we found in the wild in OCaml programs typically combine "negative" constructs (functions, lazy, records) rather than infinite lists or trees. A possible design would be to distinguish an "inductive" sub-space of recursive types whose recursive occurrences are forbidden in negative positions, and whose constructors are not given the* Guard *mode in our system. In another direction, Jeannin, Kozen, and Silva [2017] propose language extensions to make it easier to operate over cyclic structures.*

4 A MODE SYSTEM FOR RECURSIVE DEFINITIONS

4.1 Access/Usage Modes

Figure 2 defines the access/usage modes that we introduced in Section 2.1, their order structure, and the mode composition operations. The modes are as follows:

Ignore is for sub-expressions that are not used at all during the evaluation of the whole program. This is the mode of a variable in an expression in which it does not occur.

Delay means that the context can be evaluated (to a weak normal-form) without evaluating its argument. $\lambda x.\, \square$ is a delay context.

Guard means that the context returns the value as a member of a data structure, for example a variant constructor or record. $K\,(\square)$ is a guard context. The value can safely be defined mutually-recursively with its context, as in $\texttt{let rec}\ x = K\,(x)$.

Return means that the context returns its value without further inspection. This value cannot be defined mutually-recursively with its context, to avoid self-loops: in $\texttt{let rec}\ x = x$ and $\texttt{let rec}\ x = \texttt{let}\ y = x\ \texttt{in}\ y$, the rightmost occurrence of x is in Return context.

[2] In more expressive settings, the structure of usage modes does depend on the structure of values, and checks need to be presented as a refinement of a ML type system. We discuss this in Section 8. Our modes are a degenerate case, a refinement of uni-typed ML.

Modes $\ni m ::=$ Ignore | Delay | Guard | Return | Dereference

Mode order:
Ignore $\prec$ Delay $\prec$ Guard $\prec$ Return $\prec$ Dereference

Mode composition rules :

Ignore $[m]$	=	Ignore	=	m [Ignore]
Delay $[m \succ \text{Ignore}]$	=	Delay		
Guard [Return]	=	Guard		
Guard $[m \neq \text{Return}]$	=	m		
Return $[m]$	=	m		
Dereference $[m \succ \text{Ignore}]$	=	Dereference		

Mode composition as a table:

$m\,[m']$	Ignore	Delay	Guard	Return	Dereference	m
Ignore	Ignore	Ignore	Ignore	Ignore	Ignore	
Delay	Ignore	Delay	Delay	Delay	Dereference	
Guard	Ignore	Delay	Guard	Guard	Dereference	
Return	Ignore	Delay	Guard	Return	Dereference	
Dereference	Ignore	Delay	Dereference	Dereference	Dereference	
m'						

Fig. 2. Access/usage modes and operations

Dereference means that the context consumes, inspects and uses the value in arbitrary ways. Such a value must be fully defined at the point of usage; it cannot be defined mutually-recursively with its context. `match` □ `with` h is a Dereference context.

Remark 2 (Discarding). *The* Guard *mode is also used for subterms whose result is discarded by the evaluation of their context. For example, the hole* □ *is in a* Guard *context in* (`let` x = □ `in` u)*, if* x *is never used in* u*; even if the hole value is not needed, call-by-value reduction will first evaluate it and discard it. When these subterms participate in a cyclic definition, they cannot create a self-loop, so we consider them guarded.*

Our ordering $m \prec m'$ places less demanding, more permissive modes that do not involve dereferencing variables (and so permit their use in recursive definitions), below more demanding, less permissive modes.

Each mode is closely associated with particular expression contexts. For example, t □ is a Dereference context, since the function t may access its argument in arbitrary ways, while $\lambda x.$ □ is a Delay context.

Mode composition corresponds to context composition, in the sense that if an expression context $E[\square]$ uses its hole at mode m (to compute a result), and a second expression context $E'[\square]$ uses its hole at mode m', then the composition of contexts $E[E'[\square]]$ uses its hole at mode $m\,[m']$. Like context composition, mode composition is associative, but not commutative: Dereference [Delay] is Dereference, but Delay [Dereference] is Delay.

Continuing the example above, the context t $(\lambda x.\,\square)$, formed by composing t □ and $\lambda x.$ □, is a Dereference context: the intuition is that the function t may pass an argument to its input and then access the result in arbitrary ways. In contrast, the context $\lambda x.$ $(t\ \square)$, formed by composing $\lambda x.$ □

$$\boxed{\text{Term judgment } \Gamma \vdash t : m}$$

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \qquad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m\,[\text{Delay}]}{\Gamma \vdash \lambda x.\, t : m} \qquad \frac{\Gamma_t \vdash t : m\,[\text{Dereference}] \qquad \Gamma_u \vdash u : m\,[\text{Dereference}]}{\Gamma_t + \Gamma_u \vdash t\ u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m\,[\text{Guard}])^i}{\sum (\Gamma_i)^i \vdash K\ (t_i)^i : m} \qquad \frac{\Gamma_t \vdash t : m\,[\text{Dereference}] \qquad \Gamma_h \vdash^{\text{cl}} h : m}{\Gamma_t + \Gamma_h \vdash \texttt{match}\ t\ \texttt{with}\ h : m}$$

$$\frac{(x_i : \Gamma_i)^i \vdash \texttt{rec}\ b \qquad \left(m'_i\right)^i \stackrel{\text{def}}{=} (\max(m_i, \text{Guard}))^i \qquad \Gamma_u, (x_i : m_i)^i \vdash u : m}{\sum \left(m'_i\,[\Gamma_i]\right)^i + \Gamma_u \vdash \texttt{let rec}\ b\ \texttt{in}\ u : m}$$

$$\boxed{\text{Clause judgments } \Gamma \vdash^{\text{cl}} h : m \text{ and } \Gamma \vdash^{\text{cl}} p \to u : m}$$

$$\frac{\left(\Gamma_i \vdash^{\text{cl}} p_i \to u_i : m\right)^i}{\sum (\Gamma_i)^i \vdash^{\text{cl}} (p_i \to u_i)^i : m} \qquad \frac{\Gamma, (x_i : m_i)^i \vdash u : m}{\Gamma \vdash^{\text{cl}} K\ (x_i)^i \to u : m}$$

$$\boxed{\text{Binding judgment } (x_i : \Gamma_i)^i \vdash \texttt{rec}\ b}$$

$$\frac{\left(\Gamma_i, \left(x_j : m_{i,j}\right)^{j \in I} \vdash t_i : \text{Return}\right)^{i \in I} \qquad \left(m_{i,j} \preceq \text{Guard}\right)^{i,j} \qquad \left(\Gamma'_i = \Gamma_i + \sum \left(m_{i,j}\,[\Gamma'_j]\right)^j\right)^i}{\left(x_i : \Gamma'_i\right)^{i \in I} \vdash \texttt{rec}\ (x_i = t_i)^{i \in I}}$$

Fig. 3. Mode inference rules

and $t\ \square$, is a Delay context: the contents of the hole will not be touched before the abstraction is applied.

Finally, Ignore is the absorbing element of mode composition ($m\,[\text{Ignore}] = \text{Ignore} = \text{Ignore}\,[m]$), Return is an identity ($\text{Return}\,[m] = m = m\,[\text{Return}]$), and composition is idempotent ($m\,[m] = m$).

4.2 Inference Rules

Environment notations. Our environments Γ associate variables x with modes m. We write Γ_1, Γ_2 for the union of two environments with disjoint domains, and $\Gamma_1 + \Gamma_2$ for the merge of two overlapping environments, taking the maximum mode for each variable. We sometimes use family notation for environments, writing $(\Gamma_i)^i$ to indicate the disjoint union of the members, and $\sum (\Gamma_i)^i$ for the non-disjoint merge of a family of environments.

Inference rules. Figure 3 presents the inference rules for access/usage modes. The rules are composed into several different judgments, even though our simple core language makes it possible

to merge them. In the full system for OCaml the decomposition is necessary to make the system manageable.

Section 4.3 (Examples) contains examples of mode judgments in our system, corresponding to recursive definitions that are accepted or rejected. Looking at those examples in parallel may help understand some of the inference rules, in particular for `let rec`.

Variable and subsumption rules. The variable rule is as one would expect: the usage mode of x in an m-context is m. In this declarative presentation, we let the rest of the environment Γ be arbitrary; we could also have imposed that it map all variables to Ignore. Our algorithmic check returns the "least demanding" environment Γ for all satisfiable judgments, so it uses Ignore in any case.

We have a subsumption rule; for example, if we want to check t under the mode Guard, it is always sound to attempt to check it under the stronger mode Dereference. Our algorithmic check will never use this rule; it is here for completeness. The direction of the comparison may seem unusual. We can coerce a $\Gamma \vdash t : m$ into $\Gamma \vdash t : m'$ when $m \succ m'$ holds, while we might expect $m \leq m'$. This comes from the fact that our backwards reading is opposite to the usual reading direction of type judgments, and influenced our order definition. When $m \succ m'$ holds, m is *more demanding* than m', which means (in the usual subtyping sense) that it classifies *fewer* terms.

Simple rules. We have seen the $\lambda x.\, t$ rule already, in Section 2.2. Since λ delays evaluation, checking $\lambda x.\, t$ in a usage context m involves checking the body t under the weaker mode $m\,[\mathsf{Delay}]$. The necessary constraints Γ are returned, after removing the constraint over x[3].

The application rule checks both the function and its argument in a Dereference context, and merges the two resulting environments, taking the maximum (most demanding) mode on each side; a variable y is dereferenced by $t\ u$ if it is dereferenced by either t or u.

The constructor rule is similar to the application rule, except that the constructor parameters appear in Guard context, rather than Dereference.

Pattern-matching. The rule for `match` t `with` h relies on a different *clause judgment* $\Gamma \vdash^{\mathsf{cl}} h : m$ that checks each clause in turn and merges their environments. On a single clause $K\ (x_i)^i \to u$, we check the right-hand-side expressions u in the ambient mode m, and remove the pattern-bound variables $(x_i)^i$ from the environment.[4]

Recursive definitions. The rule for mutually-recursive definitions `let rec` b `in` u is split into two parts with disjoint responsibilities. First, the binding judgment $(x_i : \Gamma_i)^i \vdash \texttt{rec}\ b$ computes, for each definition $x_i = e_i$ in a recursive binding b, the usage Γ_i of the ambient context before the recursive binding — we detail its definition below.

Second, the `let rec` b `in` u rule of the term judgment takes these Γ_i and uses them under a composition $m'_i\,[\Gamma_i]$, to account for the actual usage mode of the variables. (Here $m\,[\Gamma]$ denotes the pointwise lifting of composition for each mode in Γ.) The usage mode m'_i is a combination of the usage mode in the body u and Guard, used to indicate that our call-by-value language will compute the values now, even if they are not used in u, or only under a delay — see Remark 2 (Discarding).

Deriving a simple `let` *rule.* Before we delve into the more general rule for mutually-recursive definitions, let us mention the particular case of a single, non-recursive definition `let` $x = t$ `in` u.

[3]In situations where it is desirable to have a richer mode structure to analyze function applications, as considered by some of the related work (Section 8), we could use the mode m_x in a richer return mode $m_x \to m$.

[4]If we wanted a finer-grained analysis of usage of the sub-components of our data, we would use the sub-modes $(m_i)^i$ of the pattern variables to enrich the datatype of the pattern scrutinee.

The general rule simplifies itself into the following:

$$\frac{\Gamma_t \vdash t : \mathsf{Return} \qquad m' \stackrel{\text{def}}{=} \max(m_x, \mathsf{Guard}) \qquad \Gamma_u, x : m_x \vdash u : m}{m' \left[\Gamma_t\right] + \Gamma_u \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : m}$$

Binding judgment and mutual recursion. The *binding judgment* $(x_i : \Gamma_i)^{i \in I} \vdash \mathtt{rec}\ b$ is independent of the ambient context and usage mode; it checks recursive bindings in isolation in the Return mode, and relates each name x_i introduced by the binding b to an environment Γ_i on the ambient free variables.

In the first premise, for each binding $(x_i = t_i)$ in b, we check the term t_i in a context split in two parts, some usage context Γ_i on the ambient context around the recursive definition, and a context $\left(x_j : m_{i,j}\right)^{j \in I}$ for the recursively-bound variables, where $m_{i,j}$ is the mode of use of x_j in the definition of x_i.

The second premise checks that the modes $m_{i,j}$ are Guard or less demanding, to ensure that these mutually-recursive definitions are valid. This is the check mentioned at the end of Section 2.2 (An Inference System (and Corresponding Backwards Analysis)).

The third premise makes mutual-recursion safe by turning the Γ_i into bigger contexts Γ_i' taking transitive mutual dependencies into account: if a recursive definition $x_i = e_i$ uses the mutually-defined variable x_j under the mode $m_{i,j}$, then we ask that the final environment Γ_i' for e_i contains what you need to use e_j under the mode $m_{i,j}$, that is $m_{i,j}\left[\Gamma_j'\right]$. This set of recursive equations corresponds to the fixed point of a monotone function, so in particular it has a unique least solution.

Note that because the $m_{i,j}$ must be below Guard, we can show that $m_{i,j}\left[\Gamma_j\right] \preceq \Gamma_j$. In particular, if we have a single recursive binding, we have $\Gamma_i \geq m_{i,i}\left[\Gamma_i\right]$, so the third premise is equivalent to just $\Gamma_i' \stackrel{\text{def}}{=} \Gamma_i$: the Γ_i' and Γ_i only differ for non-trivial mutual recursion.

Unique minimal environment. In Appendix A (Properties of our Typing Judgment) in the extended version we develop some direct meta-theoretic properties of our inference rules. We summarize here the key results. For each $t : m$, there exists a *minimal* environment Γ such that $\Gamma \vdash t : m$ holds.

Theorem 1 (Principal environments). *Whenever both* $\Gamma_1 \vdash t : m$ *and* $\Gamma_2 \vdash t : m$ *hold, then* $\min(\Gamma_1, \Gamma_2) \vdash t : m$ *also holds.*

We also define minimal *derivations*, which restrict the non-determinism in the variable and binding rules, and the way subsumption may be used. Minimal derivations have minimal environments and a syntax-directed structure. They precisely characterize the behavior of our algorithm, implemented in the OCaml compiler: given $t : m$, it constructs a minimal derivation of the form $\Gamma \vdash t : m$, and returns the environment Γ, which is minimal for $t : m$.

4.3 Examples

Our checker accepts a definition $\mathtt{let\ rec}\ x = t$ if there exists a mode judgment $\Gamma \vdash t : \mathsf{Return}$ that assigns a mode to x in Γ that is Guard or smaller. The definition is rejected if the mode of x in the minimal judgment is Return or Dereference. Let us discuss various examples of definitions that are accepted or rejected, along with the corresponding minimal judgments.

Separating Return *from* Guard, Delay. The definitions $\mathtt{let\ rec}\ x = \mathtt{Fix}\ x$ or $\mathtt{let\ rec}\ f = \lambda x.\ f\ x$ are valid: the definitions admit the mode judgments $x : \mathsf{Guard} \vdash \mathtt{Fix}\ x : \mathsf{Return}$ and $f : \mathsf{Delay} \vdash \lambda x.\ f\ x : \mathsf{Return}$. On the other hand, the definition $\mathtt{let\ rec}\ x = x$ is invalid, as the best judgment for its body is $x : \mathsf{Return} \vdash x : \mathsf{Return}$, with $x : \mathsf{Return}$ (stricter than Guard) in Γ.

Separating Guard *from* Delay. In the valid example $\texttt{let rec}\ f = \lambda x.\, f\ x$, the subterm $f\ x$ is dereferencing f; the mode of f in the outer environment is still Delay thanks to the composition Delay [Dereference] = Delay. On the other hand, if we had moved the application outside the delay, $\texttt{let rec}\ f = (\lambda x.\, f)\ u$, this definition would be rejected, as we would have f : Dereference thanks to the composition Dereference [Delay] = Dereference.

Guard behaves differently than Delay: our constructors are strict, so dereferencing inside a guard is also a dereference. For example, considering a function g defined outside, both $\texttt{let rec}\ x = g\ (\texttt{Fix}\, x)$ and $\texttt{let rec}\ x = \texttt{Fix}\ (g\ x)$ are rejected, as the mode of x is Dereference in the right-hand side of the definition. In the first case this mode comes from the composition Dereference [Guard] = Dereference, in the second case from Guard [Dereference] = Dereference.

Separating Delay *from* Ignore. Notice that if we have x : Delay $\vdash t[x]$: Return then we have x : Dereference, g : Dereference $\vdash g\ (t[x])$: Return but if we have x : Ignore $\vdash t[x]$: Return then we have x : Ignore, g : Dereference $\vdash g\ (t[x])$: Return. So a declaration of the form $\texttt{let rec}\ x = g\ (t[x])$ is rejected if t uses the variable x at mode Delay, but accepted if x is not used (x : Ignore).

Separating Return *from* Dereference. Similarly, if we have x : Return $\vdash t[x]$: Return (for example $t[x]$ is x or $(\texttt{let}\ y = x\ \texttt{in}\ y)$), then we have x : Guard $\vdash\ \texttt{Fix}\ (t[x])$: Return, but if we have x : Dereference $\vdash t[x]$: Return then we have x : Dereference in the environment of $\texttt{Fix}\ (t[x])$. In particular, $\texttt{let rec}\ x = \texttt{Fix}\ (t[x])$ is accepted in the first case and rejected in the second.

Simple `let` *examples.* (These examples are easier to follow by using the simple `let` rule than the general `let rec` rule.). What is the mode x in $\texttt{let}\ z = \lambda y.\, \texttt{Pair}\ (x, y)\ \texttt{in}\ \texttt{Fix}\, z$? The mode of x in the definition $\lambda y.\, \texttt{Pair}\ (x, y)$ is Delay, and the mode of z in the body $\texttt{Fix}\, z$ is Guard. The final context (at global mode Return) is Guard [x : Delay], that is x : Delay.

In the case of $\texttt{let}\ z = \lambda y.\, \texttt{Pair}\ (x, y)\ \texttt{in}\ g\ z$, the final context is Dereference [x : Delay] = x : Dereference. Finally, the premise $m' = \max(m, \text{Guard})$ of the rule comes into play when the body delays or ignores the defined variable: in the case of $\texttt{let}\ z = g\ x\ \texttt{in}\ y$, the mode of z in y is Ignore, but the mode of x in the whole term is not Ignore [x : Dereference], which would be x : Ignore, but rather max(Ignore, Guard) [x : Dereference], which is x : Dereference.

`let rec` *examples.* The delicate aspect of the $(x_i : \Gamma_i)^i \vdash \texttt{rec}\ (x_i = t_i)^i$ judgment is the fixpoint of equations $\Gamma_i' = \Gamma_i + \sum \left(m_{i,j} \left[\Gamma_j'\right]\right)^j$, which computes a "transitive closure" of usage modes of the mutually-recursively-defined variables. Consider for example the term t defined as

$$t \quad \stackrel{\text{def}}{=} \quad \texttt{let rec}\ x' = x\ \texttt{and}\ y = \texttt{Fix}\ (x')\ \texttt{and}\ z = g\ y\ \texttt{in}\ z$$

We index the three definitions by $i \in I$ with $I \stackrel{\text{def}}{=} \{x', y, z\}$. The contexts $(\Gamma_i)^i$ of the rule correspond to the dependency of each right-hand-side on non-mutually-recursive variables. The $\left(\Gamma_i'\right)^i$ are defined by a system of recursive equations, depending on each Γ_j and the mode of use of the variable j in the definition of i. We have:

$$\begin{array}{ll}
\Gamma_{x'} \stackrel{\text{def}}{=} (x : \text{Return}) & \Gamma_{x'}' \stackrel{\text{def}}{=} \Gamma_x \\
\Gamma_y \stackrel{\text{def}}{=} \emptyset & \Gamma_y' \stackrel{\text{def}}{=} \Gamma_y + \text{Guard}\left[\Gamma_{x'}'\right] \\
\Gamma_z \stackrel{\text{def}}{=} (g : \text{Dereference}) & \Gamma_z' \stackrel{\text{def}}{=} \Gamma_z' + \text{Dereference}\left[\Gamma_y'\right]
\end{array}$$

The smallest fixpoint solution has $\Gamma_{x'}' = (x : \text{Return})$, $\Gamma_y' = (x : \text{Guard})$, and $\Gamma_z' = (g : \text{Dereference}, x : \text{Dereference})$. In particular, notice how x is accessed at mode Dereference by z, even though it does not syntactically appear in its definition. The whole term t, in the ambient mode Return, is in the environment $\Gamma_z' = (g : \text{Dereference}, x : \text{Dereference})$. If we had used a simpler `rec` rule that would return the $(\Gamma_i)^i$ instead of the $\left(\Gamma_i'\right)^i$, immediate usage rather than

transitive usage, t would typed in the environment $\Gamma_z = g : \mathsf{Dereference}$, that is with $x : \mathsf{Ignore}$. This would be unsound, for example the vicious definition $\texttt{let rec}\ x = t$ would be accepted.

4.4 Discussion

Declarative vs. algorithmic rules. A presentation of a type system can be more "algorithmic" or more "declarative", sitting on a continuous spectrum. More-declarative systems have convenient typing rules corresponding to reasoning principles that are sound in the metatheory, but may be harder to implement in practice. More-algorithmic systems have more rigid inference rules, that are easier to implement as a checking or inference system (typically they may be syntax-directed); some valid reasoning principles may not be available as rules but only as admissible rules (requiring a global rewrite of the derivation) or not at all.

In our system, the variable and subsumption rules are typically declarative: the variable rule has an undetermined Γ context, and the subsumption rule makes the system non-syntax-directed. Without those two rules, it would not be possible to prove $x : \mathsf{Guard}, y : \mathsf{Dereference} \vdash y : \mathsf{Return}$, but only the stronger judgment $x : \mathsf{Ignore}, y : \mathsf{Return} \vdash y : \mathsf{Return}$.

On the other hand, our binding rule has an algorithmic flavor: it introduces a family of contexts $\left(\Gamma'_i\right)^i$ that is uniquely determined as the solution of a system of recursive equations $\left(\Gamma'_i = \Gamma_i + \sum \left(m_{i,j} \left[\Gamma'_j\right]\right)^j\right)^i$, so its application requires computing a fixpoint. A more declarative presentation would allow "guessing" any family $(\Gamma_i)^i$ that satisfies the inequations necessary for soundness:

$$\frac{\left(\Gamma_i, \left(x_j : m_{i,j}\right)^{j \in I} \vdash t_i : \mathsf{Return}\right)^{i \in I} \qquad \left(m_{i,j} \preceq \mathsf{Guard}\right)^{i,j} \qquad \left(\Gamma_i \geq \Gamma_i + \sum \left(m_{i,j} \left[\Gamma_j\right]\right)^j\right)^i}{\left(x_i : \Gamma_i\right)^{i \in I} \vdash \mathsf{rec}\ \left(x_i = t_i\right)^{i \in I}}$$

Backwards type systems. Typing rules are a specialized declarative language to describe and justify various computational processes related to a type system (type checking, type inference, elaboration, etc.). Our mode system read "backwards" is one possible way to describe the static analysis we are capturing, which could also be described in many other ways: as pseudocode, as a fixpoint of equations, through a denotational semantics, etc. In general we believe that reading type systems backwards can give a nice, compact, declarative presentation of certain demand analyses, in a language that type designers are already familiar with.

Our terminology follows the "backward analysis" notion described for logic programming languages by Genaim and Codish [2001], i.e. our algorithm answers the question (posed in that work) "Given a program and an assertion at a given program point, what are the weakest requirements on the inputs to the program which guarantee that the assertion will hold whenever execution reaches that point?" For our analysis, the *assertion* is the mode under which an expression is checked, and the *requirements on the inputs* correspond to the computed environment Γ.

Backward analyses for functional languages also appear in work by Hughes [Hughes 1987]. A notable difference is that Hughes-style "demand analyses"[5] are typically presented in a denotational style, using tools of domain theory. However, some recent work (such as the cardinality analysis by Sergey et al. [2017b]) presents backward analyses in a syntactic style more similar to that used in the present paper.

[5]For a recent example, see the unpublished draft Sergey, Peyton-Jones, and Vytiniotis [2017a]. Thanks are due to Joachim Breitner for the reference.

Modes as modalities. Untyped or dynamically-typed languages can be seen as "uni-typed", with a "universal type" U of all values. Language constructions can be presented as section/retraction pairs from U to a type that computes, such as $\mathsf{U} \to \mathsf{U}$ for functions or $\mathsf{U} \times \mathsf{U} \times \cdots \times \mathsf{U}$ for tuples; for example, the untyped term $(\lambda x.\, t)\ u$ can be explicitated into $\mathsf{app}\ (\mathsf{lam}(\lambda x.\, t))\ u$, for combinators $\mathsf{app} : \mathsf{U} \to (\mathsf{U} \to \mathsf{U})$ and $\mathsf{lam} : (\mathsf{U} \to \mathsf{U}) \to \mathsf{U}$ such that $\mathsf{app} \circ \mathsf{lam}$ is the identity on $\mathsf{U} \to \mathsf{U}$ — but $\mathsf{lam} \circ \mathsf{app}$ gets stuck on non-functions.

Rather than *types*, it is more precise to see our modes as *modalities* on this universal type U. The hypothesis $x : m$ in a context would be a modal hypothesis $x :^m \mathsf{U}$, and the section combinators are given modal types. Within the modal framework of Abel and Bernardy [2020] for example, writing $(m : A) \to B$ for the modal function type, some of our typing rules could be modelled with $\mathsf{lam} : (\mathsf{Delay} : (\mathsf{Dereference} : \mathsf{U}) \to \mathsf{U}) \to \mathsf{U}$, $\mathsf{app} : (\mathsf{Dereference} : \mathsf{U}) \to (\mathsf{Dereference} : \mathsf{U}) \to \mathsf{U}$, and for datatypes something like $\mathsf{pack}^K_d : (\mathsf{Guard} : \mathsf{U}) \to^d \mathsf{U}$, and $\mathsf{unpack}^K_d : (\mathsf{Dereference} : \mathsf{U}) \to (\mathsf{U} + \mathsf{U}^d)$, where K is a constructor of arity d, the U+ return value of unpacking indicates an incompatible constructor, and U^d is $\mathsf{U} \times \mathsf{U} \times \ldots$, d times.

This view naturally extends to supporting finer-grained analyses and abstraction of the sort found in other work (e.g. the system introduced by Dreyer [2004]). In our system every function argument has mode Dereference; a refinement that allowed types and modes to interact could support a range of modes for functions that used their arguments in different ways. However, this additional expressivity would require substantial changes to OCaml's type system, and a proper treatment of abstraction would require some form of mode polymorphism. (For example, in the appliction function `let h g x = g x`, mode polymorphism is required to support applying the `h` to all possible functions `g`, some of which dereference their argument, and some of which do not.)

Our aim of replacing OCaml's existing syntactic check with a more principled version way did not justify this substantial additional complexity. More generally, our view is that abstraction over modes is more suited to module systems (for which the system described by Dreyer [2004] was developed), where types are already explicit, than to the term languages that our system is designed for. This view is supported by the fact that, as far as we know, although existing languages support a variety of checks on well-formedness of recursive definitions, there has been (as far as we know) no attempt in any of these to incorporate a system with support for mode abstraction.

5 META-THEORY: SOUNDNESS

5.1 Operational Semantics

Figure 4 presents our operational semantics, largely reused from Nordlander, Carlsson, and Gill [2008] with extensions (support for algebraic datatypes) and changes (use of *reduction at a distance*). Unless explicitly noted, the content and ideas in this Subsec 5.1 come from their work.

Weak values. As we have seen, constructors in recursive definitions can be used to construct cyclic values. For example, the definition $\texttt{let rec}\ x = \texttt{Cons}\ (\texttt{One}\ (\emptyset), x)$ is normal for this reduction semantics. The occurrence of the variable x inside the `Cons` cell corresponds to a back-reference, the cell address in a cyclic in-memory representation.

This key property is achieved by defining a class of *weak values*, noted w, to be either (strict) values or variables. Weak values occur in the definition of the semantics wherever a cyclic reference can be passed without having to dereference.

Several previous works (see Section 8 (Related Work)) defined semantics where β-redexes have the form $(\lambda x.\, t)\ w$, to allow yet-unevaluated recursive definitions to be passed as function arguments. OCaml does not allow this (a function call requires a fully-evaluated argument), so our redexes are the traditional $(\lambda x.\, t)\ v$. This is a difference from Nordlander, Carlsson, and Gill [2008]. On the other

$$\begin{array}{rcl} \text{Values} \ni v & ::= & \lambda x.\, t \mid K\,(w_i)^i \mid L[v] \\ \text{WeakValues} \ni w & ::= & x, y, z \mid v \mid L[w] \\ \\ \text{ValueBindings} \ni B & ::= & (x_i = v_i)^i \\ \text{BindingCtx} \ni L & ::= & \Box \mid \texttt{let rec}\ B\ \texttt{in}\ L \end{array} \qquad \begin{array}{rcl} \text{EvalCtx} \ni E & ::= & \Box \mid E[F] \\ \text{EvalFrame} \ni F & ::= & \Box\, t \mid t\, \Box \\ & & \mid K\,((t_i)^i, \Box, (t_j)^j) \\ & & \mid \texttt{match}\ \Box\ \texttt{with}\ h \\ & & \mid \texttt{let rec}\ b, x = \Box, b'\ \texttt{in}\ u \\ & & \mid \texttt{let rec}\ B\ \texttt{in}\ \Box \end{array}$$

$$\frac{}{L[\lambda x.\, t]\ v \to^{\text{hd}} L[t[v/x]]} \qquad \frac{\forall (K'\left(x'_j\right)^j \to u') \in h,\ K \neq K'}{\texttt{match}\ L[K\,(w_i)^i]\ \texttt{with}\ (h \mid K\,(x_i)^i \to u \mid h') \to^{\text{hd}} L[u[w_i/x_i]^i]}$$

$$\frac{t \to^{\text{hd}} t'}{E[t] \to E[t']} \qquad \frac{(x = v) \overset{\text{ctx}}{\in} E}{E[x] \to E[v]} \qquad \frac{(x = v) \overset{\text{frame}}{\in} F \ \vee\ (x = v) \overset{\text{ctx}}{\in} E}{(x = v) \overset{\text{ctx}}{\in} E[F]}$$

$$\frac{(x = v) \in B}{(x = v) \overset{\text{frame}}{\in} \texttt{let rec}\ B\ \texttt{in}\ \Box} \qquad \frac{(x = v) \in (b \cup b')}{(x = v) \overset{\text{frame}}{\in} \texttt{let rec}\ b, y = \Box, b'\ \texttt{in}\ u}$$

Fig. 4. Operational semantics

hand, we do allow cyclic datatype values by only requiring weak values under data constructors: the corresponding value form is $K\,(w_i)^i$.

Bindings in evaluation contexts. An *evaluation context* E is a stack of *evaluation frames* F under which evaluation may occur. Our semantics is under-constrained (for example, $t\ u$ may perform reductions on either t or u), as OCaml has unspecified evaluation order for applications and constructors, but making it deterministic would not change much.

One common aspect of most operational semantics for **`let rec`**, ours included, is that $\texttt{let rec}\ B\ \texttt{in}\ \Box$ can be part of evaluation contexts, where B represents a recursive "value binding", an island of recursive definitions that have all been reduced to values. This is different from traditional source-level operational semantics of $\texttt{let}\ x = v\ \texttt{in}\ u$, which is reduced to $u[v/x]$ before going further. In `letrec` blocks this substitution reduction is not valid, since the value v may refer to the name x, and so instead "value bindings" remain in the context, in the style of explicit substitution calculi. We call these context fragments "binding contexts" L.

Head reduction. Head redexes, the sources of the head-reduction relation $t \to^{\text{hd}} t'$, come from applying a λ-abstraction or from pattern-matching on a head constructor. Following ML semantics, pattern-matching is ordered: only the first matching clause is taken.

One mildly original feature of our head reduction is the use of *reduction at a distance*, where binding contexts L are allowed to be presented in the middle of redexes, and lifted out of the reduced term. This presentation is common in explicit-substitution calculi[6], as it gives the minimal amount of lifting of explicit substitutions required to avoid blocking reduction. In the calculus of Nordlander, Carlsson, and Gill [2008], lifting was permitted in arbitrary positions by the Merge rule. For example, the reduction sequence $(\texttt{let rec}\ B\ \texttt{in}\ \lambda x.\, t)\ v \to^* \texttt{let rec}\ B\ \texttt{in}\ t[u/x]$ is admissible

[6]See for example Accattoli and Kesner [2010], which links to earlier references on the technique.

in both systems, but the "useless" reduction $(\texttt{let rec}\ B\ \texttt{in}\ x)\ v \to^* \texttt{let rec}\ B\ \texttt{in}\ x\ v$ is not present in our system. Reduction at a distance tends to make definitions crisper and simplify proofs.[7]

Reduction. Reduction $t \to t'$ may happen under any evaluation context. The first reduction rule is standard: any redex $H[v]$ can be reduced under an evaluation context E.

The second rule reduces a variable x in an evaluation context E by binding lookup: it is replaced by the value of the recursive binding B in the context E which defines it. This uses the auxiliary definition $(x = v) \overset{\text{ctx}}{\in} E$ to perform this lookup.

The lookup rule has worrying consequences for our rewriting relation: it makes it nondeterministic and non-terminating. Indeed, consider a weak value of the form $K\ (x)$ used, for example, in a pattern-matching $\texttt{match}\ K\ (x)\ \texttt{with}\ h$. It is possible to reduce the pattern-matching immediately, or to first lookup the value of x and then reduce. Furthermore, it could be the case that x is precisely defined by a cyclic binding $x = K\ (x)$. Then the lookup rule would reduce to $\texttt{match}\ K\ (K\ (x))\ \texttt{with}\ h$, and we could keep looking indefinitely. Nordlander, Carlsson, and Gill [2008] discuss this in detail and prove that the reduction is in fact confluent modulo unfolding. (Allowing these irritating but innocuous behaviors is a large part of what makes their semantics simpler than previous presentations.)

Example. Consider the following program:

$$\begin{array}{l}\texttt{let rec}\ \infty = (\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ x)\ \texttt{in}\\ \quad \texttt{match}\ \infty\ \texttt{with}\ (\mathsf{Z} \to \mathsf{None} \mid \mathsf{S}\,y \to \mathsf{Some}\,y)\end{array}$$

The first binding $\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ x$ is not a value yet, only a weak value. The first reduction this program can take is to lookup the right-hand-side occurence of x:

$$\begin{array}{l}\to \texttt{let rec}\ \infty = (\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ \mathsf{S}\,x)\ \texttt{in}\\ \quad \texttt{match}\ \infty\ \texttt{with}\ (\mathsf{Z} \to \mathsf{None} \mid \mathsf{S}\,y \to \mathsf{Some}\,y)\end{array}$$

After this reduction ∞, is bound to a value, so it can in turn be looked up in $\texttt{match}\ \infty$:

$$\begin{array}{l}\to \texttt{let rec}\ \infty = (\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ \mathsf{S}\,x)\ \texttt{in}\\ \quad \texttt{match}\ (\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ \mathsf{S}\,x)\ \texttt{with}\\ \quad (\mathsf{Z} \to \mathsf{None} \mid \mathsf{S}\,y \to \mathsf{Some}\,y)\end{array}$$

At this point we have a match redex of the form $\texttt{match}\ L[\mathsf{S}\,x]$, which gets reduced by lifting the binding context L:

$$\begin{array}{l}\to \texttt{let rec}\ \infty = (\texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ \mathsf{S}\,x)\ \texttt{in}\\ \quad \texttt{let rec}\ x = \mathsf{S}\,x\ \texttt{in}\ \mathsf{Some}\,x\end{array}$$

5.2 Failures

In this section, we are interested in formally defining dynamic failures. When can we say that a term is "wrong"? — in particular, when is a valid implementation of the operational semantics allowed to crash? This aspect is not discussed in detail by Nordlander, Carlsson, and Gill [2008], so we had to make our own definitions; we found it surprisingly subtle.

The first obvious sort of failure is a type mismatch between a value constructor and a value destructor: application of a non-function, pattern-matching on a function instead of a head constructor, or not having a given head constructor covered by the match clauses. These failures would be ruled out by a simple type system and exhaustivity check.

[7] For an example of beneficial use of reduction-at-distance in previous work from the rewriting community, see the at-a-distance presentation of the π-calculus in Accattoli [2013].

The more challenging task is defining failures that occur when trying to access a recursively-defined variable too early. The lookup reduction rule for a term $E[x]$ looks for the value of x in a binding of the context E. This value may not exist (yet), and that may or may not represent a runtime failure.

We assume that bound names are all distinct, so there may not be several v values. The only binders that we reduce under are **`let rec`**, so x must come from one; however, it is possible that x is part of a **`let rec`** block currently being evaluated, with an evaluation context of the form $E[\texttt{let rec}\ (x = t, y = E')\ \texttt{in}\ u]$ for example, and that x's binding has not yet been reduced to a value.

However, in the presence of data constructors that permit building cyclic values not all such cases are failures. For example the term $\texttt{let rec}\ x = \texttt{Pair}\ (x, t)\ \texttt{in}\ x$ can be decomposed into $E[x]$ to isolate the occurrence of x as the first member of the pair. This occurrence of x is in reducible position, but there is no v such that $(x = v) \overset{\text{ctx}}{\in} E$, unless t is already a weak value.

To characterize failures during recursive evaluation, we propose to restrict ourselves to *forcing contexts*, denoted E_f, that must access or return the value of their hole. A variable in a forcing context that cannot be looked up in the context is a dynamic failure: we are forcing the value of a variable that has not yet been evaluated. If a term contains such a variable in lookup position, we call it a *vicious* term. Figure 5 gives a precise definition of these failure terms.

$$\begin{array}{rrcl} \mathsf{HeadFrame} \ni H & ::= & \square\ v \mid \texttt{match}\ \square\ \texttt{with}\ h & \\ \mathsf{ForcingFrame} \ni F_\mathsf{f} & ::= & \square\ v \mid v\ \square \mid \texttt{match}\ \square\ \texttt{with}\ h & \mathsf{Mismatch} \overset{\text{def}}{=} \{E[H[v]] \mid H[v] \not\twoheadrightarrow^{\mathsf{hd}}\} \\ \mathsf{ForcingCtx} \ni E_\mathsf{f} & ::= & L \mid E[F_\mathsf{f}[L]] & \end{array}$$

$$\mathsf{Vicious} \overset{\text{def}}{=} \{E_\mathsf{f}[x] \mid \nexists v, (x = v) \overset{\text{ctx}}{\in} E_\mathsf{f}\}$$

Fig. 5. Failure terms

Mismatches are characterized by *head frames*, context fragments that would form a β-redex if filled with a value of the correct type. A term of the form $H[v]$ that is stuck for head-reduction is a constructor-destructor mismatch.

The definition of forcing contexts E_f takes into account the fact that recursive value bindings remain, floating around, in the evaluation context. A forcing frame F_f is a context fragment that forces evaluation of its variable; it would be tempting to say that a forcing context is necessarily of the form $\square$ or $E[F_\mathsf{f}]$, but for example $F_\mathsf{f}[\texttt{let rec}\ B\ \texttt{in}\ \square]$ must also be considered a forcing context.

Note that, due to the flexibility we gave to the evaluation order, mismatches and vicious terms need not be stuck: they may have other reducible positions in their evaluation context. In fact, a vicious term failing on a variable x may reduce to a non-vicious term if the binding of x is reduced to a value.

5.3 Soundness

The proofs of these results are in Appendix B in the extended version.

Lemma 1 (Forcing modes). *If* $\Gamma, x : m_x \vdash E_\mathsf{f}[x] : m$ *with* $m \geq \mathsf{Return}$, *then also* $m_x \geq \mathsf{Return}$.

Theorem 2 (Vicious). $\emptyset \vdash t : \mathsf{Return}$ *never holds for* $t \in \mathsf{Vicious}$.

Theorem 3 (Subject reduction). *If* $\Gamma \vdash t : m$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : m$.

Corollary 1. Return*-typed programs cannot go vicious.*

6 GLOBAL STORE SEMANTICS

In Section 5.1, we developed an operational semantics for `letrec` using explicit substitutions as "local stores" for recursive values. This technique was first studied in more theoretical research communities (pure lambda-calculus and rewriting), and imported in more applied programming-language works in the 1990s [Felleisen and Hieb 1992] and in particular local-store presentations of call-by-need and `letrec` [Ariola and Felleisen 1997].

Before local-store semantics were proposed, recursive values (and lazy thunks) were modelled using a "global store" semantics, more closely modelling the dummy-initialization-then-backpatching approach used in real-world implementations.

Explicit-substitution, local-store semantics enjoy at least the following advantages over the global-store semantics of `letrec`.

- Our explicit-substitution semantics is defined directly at the level of our term syntax, without going through additional features (global store, initialization and setting of memory locations) that are not part of the source language. Source-level semantics are typically more high-level and more declarative than lower-level semantics; they allow to reason on the source directly, without going through an encoding layer.
  We can compare `letrec` to the problem of defining "tail calls". One can teach tail-calls and reason about them through a compilation to a machine with an explicit call stack. But there is a source-level explanation of tail-calls, by reasoning on the size (and the growth) of the evaluation context; this explanation is simpler and makes tail-calls easier to reason about.
- Our local-store semantics allows more local reasoning: reducing a subterm in evaluation position only affects this subterm (which may contain explicit substitutions), instead of also affecting a global store. This makes it easier to define program-equivalence relations that are congruence (are stable under contexts), and generally to prove program-equivalence results.

  Example 1 (Store duplication). *For example, consider the two following programs, manipulating infinite lists of ones:*

  $$\texttt{let } o = (\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in } x) \texttt{ in } f\ o\ o$$

  $$f\ (\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in } x)\ (\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in } x)$$

  With our explicit-substitution semantics, it is trivial to show that these two terms are equivalent: they reduce to the same term, namely

  $$f\ (\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in Cons}(1, x))\ (\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in Cons}(1, x))$$

  (Note that $(\texttt{let rec } x = \texttt{Cons}(1, x) \texttt{ in } x)$ *is not a value in our semantics, only a weak value; one needs to lookup* x *once to get a value so that the* o*-binding can be reduced.)*
  In a global-store semantics, on the contrary, it is not at all obvious that the two programs are equivalent; indeed, they reduce to configurations of the following forms:

  $$([x \mapsto \texttt{Cons}(1, x)], f\ x\ x) \qquad ([x_1 \mapsto \texttt{Cons}(1, x_1), x_2 \mapsto \texttt{Cons}(1, x_2)], f\ x_1\ x_2)$$

  These configurations are not equivalent by reduction; the equational theory would need a stronger equivalence principle that could de-duplicate bisimilar fragments of the store.

However, since a large part of our community is unfamiliar with local-store semantics, in the interest of accessibility, we also propose in this section a global-store semantics for our language. We show that the soundness result for our analysis can be lifted to this semantics: Return-typed programs cannot go vicious in the global-store semantics either. This is done by formalizing a compilation pass from the local-store language to the global-store+backpatching language, and

showing a backward simulation result — effectively redoing the compilation-correctness work of Hirschowitz, Leroy, and Wells [2003, 2009] in our setting.

6.1 Target Language

The global-store language is called the "target" language as we will prove the correctness of a compilation pass from the "source" (local-store) to "target" (global-store) languages. Its grammar and operational semantics are given in Figure 6.

A store location is created uninitialized with the $\texttt{new}\ x\ \texttt{in}\ t$ constructor, and can be defined exactly once by the assignment expression $x \leftarrow t$; the heap binds locations to "heap blocks" that are either uninitialized ($\bot$) or a value. The write-once discipline is enforced by the $x \leftarrow t$ reduction rule, which requires the heap block to be $\bot$. Trying to read an uninitialized memory location is the dynamic error that our analysis prevents: it is the target equivalent of vicious terms, and we call it a "segfault" by analogy with the unpleasant consequences of the corresponding error in most compiled languages.

The reduction uses a distinguished data constructor, `Done`, playing the role of a unit value in our untyped semantics: it is the value returned by the evaluation of an assignment expression $x \leftarrow v$.

Remark 3 (Variables as locations). *We use variables as heap locations, an idea that goes back at least to [Launchbury 1993]. This is equivalent to having a distinguished syntactic category of locations thanks to the Barendregt convention — we work modulo α-equivalence and can thus always assume that bound variables are distinct from free variables of the same term. Picking a "fresh enough" variable x each time we encounter a* `new` *binder gives a new memory location. In particular, those variables do not correspond to static binding positions in the program we started computing; each time a function is called, the* `new` *binders in its body bind over α-equivalent fresh names. This technical choice will make it easier to relate to the source language, where recursive values are bound to variables in scope.*

6.2 Parallel or Order-Independent Evaluation

In Section 6.1 we chose a distinguished `Done` constructor to represent a unit value in our target language. We now choose another distinguished constructor $\texttt{Par}\ (t_1, \ldots, t_n)$ to represent a product of values to evaluate in an arbitrary order. We use a single constructor at arbitrary arities, but we could just as well use a family of constructors $\texttt{Par}_n$.

Definition 1 (Sequential and parallel evaluation).
We define the following syntactic sugar:

$$\begin{array}{lcl} (t; u) & \stackrel{\text{def}}{=} & (\texttt{match}\ t\ \texttt{with}\ (\texttt{Done} \to u)) \\ \texttt{par}\ (t_i)^{i \in I} & \stackrel{\text{def}}{=} & (\texttt{match}\ \texttt{Par}\ (t_i)^{i \in I}\ \texttt{with}\ \texttt{Par}\ (x_i)^{i \in I} \to \texttt{Done}) \end{array}$$

The term $\texttt{par}(t_i)^i$ represents the evaluation of a family of `Done`-returning term, in an arbitrary order. In particular, $(\Box; u)$ and $\texttt{par}((t_i)^{i \in I}, \Box, (t_j)^{j \in J})$ are evaluation contexts, and the following reduction rules are derivable:

$$(\emptyset, (\texttt{Done}; u)) \to^{\text{hd}} (\emptyset, u) \qquad (\emptyset, \texttt{par}(\texttt{Done})^{i \in I}) \to^{\text{hd}} (\emptyset, \texttt{Done})$$

6.3 Translation: Compiling `letrec` into Backpatching

We define in Figure 7 our translation $[\![t]\!]$ from source to target terms, explaining recursive value bindings in terms of backpatching. The interesting case is $[\![\texttt{let rec}\ b\ \texttt{in}\ u]\!]$, the others are just a direct mapping on all subterms.

$$[\![\texttt{let rec}\ (x_i = t_i)^i\ \texttt{in}\ u]\!] \qquad \stackrel{\text{def}}{=} \qquad \texttt{new}\ (x_i)^i\ \texttt{in}\ \texttt{par}\big(x_i \leftarrow [\![t_i]\!]\big)^i; [\![u]\!]$$

$$
\begin{array}{rcl}
\text{TTerms} \ni t, u & ::= & x, y, z \\
& & \mid \lambda x.\, t \mid t\ u \\
& & \mid K\ (t_i)^i \mid \texttt{match}\ t\ \texttt{with}\ h \\
& & \mid \texttt{new}\ x\ \texttt{in}\ t \\
& & \mid x \leftarrow t
\end{array}
\qquad \text{Handlers, patterns: as in the source language}
$$

$$
\begin{array}{rcl}
\text{TValues} \ni v & ::= & \lambda x.\, t \mid K\ (w_i)^i \\
\text{TWeakValues} \ni w & ::= & x, y, z \mid v
\end{array}
\qquad
\begin{array}{rcl}
\text{ValueHeaps} \ni B & ::= & \emptyset \mid B[x \mapsto v] \\
\text{Heaps} \ni H & ::= & \emptyset \mid H[x \mapsto v?] \\
\text{HeapBlocks} \ni v? & ::= & v \mid \bot
\end{array}
$$

$$
\begin{array}{rcl}
\text{TEvalCtx} \ni E & ::= & \Box \mid E[F] \\
\text{TEvalFrame} \ni F & ::= & \Box\ t \mid t\ \Box \\
& & \mid K\ ((t_i)^i, \Box, (t_j)^j) \\
& & \mid \texttt{match}\ \Box\ \texttt{with}\ h \\
& & \mid x \leftarrow \Box
\end{array}
\qquad
\begin{array}{rcl}
\text{TFEvalCtx} \ni E_f & ::= & \Box \mid E[F_f] \\
\text{TForcingFrame} \ni F_f & ::= & \Box\ t \mid t\ \Box \\
& & \mid \texttt{match}\ \Box\ \texttt{with}\ h
\end{array}
$$

CTX

$$\frac{(H_\Box, t_h) \to^{\text{hd}} (H'_\Box, t'_h)}{(H_E \uplus H_\Box, E[t_h]) \to (H_E \uplus H'_\Box, E[t'_h])}$$

$$\frac{}{(\emptyset, (\lambda x.\, t)\ u) \to^{\text{hd}} (\emptyset, t[u/x])}$$

$$\frac{\forall (K' \left(x'_j\right)^j \to u') \in h,\ K \neq K'}{(\emptyset, \texttt{match}\ K\ (w_i)^i\ \texttt{with}\ (h \mid K\ (x_i)^i \to u \mid h')) \to^{\text{hd}} (\emptyset, u[w_i/x_i]^i)}$$

NEW

$$\frac{}{(\emptyset, \texttt{new}\ x\ \texttt{in}\ t) \to^{\text{hd}} ([x \mapsto \bot], t)}$$

SET

$$\frac{}{([x \mapsto \bot], x \leftarrow v) \to^{\text{hd}} ([x \mapsto v], \text{Done})}$$

LOOKUP

$$\frac{}{([x \mapsto v], x) \to^{\text{hd}} ([x \mapsto v], v)}$$

$$\text{Segfault} \stackrel{\text{def}}{=} (H[x \mapsto \bot], E_f[x])$$

Fig. 6. Syntax and reduction of the global-store (target) language

To compile $\texttt{let rec}\ (x_i = t_i)^i\ \texttt{in}\ u$, we create uninitialized store cells for each x_i,[8] then we compute the assignments $x_i \leftarrow [\![t_i]\!]$ in an arbitrary order, and finally we evaluate $[\![u]\!]$. Note that all the x_i are in the scope of each $[\![t_j]\!]$: the translation respects the scoping of the $\texttt{let rec}\ (x_i = t_i)^i$ construct.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!] & \stackrel{\text{def}}{=} & x & [\![t\ u]\!] & \stackrel{\text{def}}{=} & [\![t]\!]\ [\![u]\!] \\
[\![\lambda x.\, t]\!] & \stackrel{\text{def}}{=} & \lambda x.\, [\![t]\!] & [\![K\ (t_i)^i]\!] & \stackrel{\text{def}}{=} & K\ \left([\![t_i]\!]\right)^i
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{match}\ t\ \texttt{with}\ (p_i \to t_i)^i]\!] & \stackrel{\text{def}}{=} & \texttt{match}\ [\![t]\!]\ \texttt{with}\ \left(p_i \to [\![t_i]\!]\right)^i \\
[\![\texttt{let rec}\ (x_i = t_i)^i\ \texttt{in}\ u]\!] & \stackrel{\text{def}}{=} & \texttt{new}\ (x_i)^i\ \texttt{in}\ \texttt{par}\left(x_i \leftarrow [\![t_i]\!]\right)^i; [\![u]\!]
\end{array}
$$

Fig. 7. Compiling letrec into store updates

[8] $(\texttt{new}\ (x_i)^{i \in I}\ \texttt{in}\ \Box)$ denotes a sequence of $(\texttt{new}\ x_i\ \texttt{in}\ \Box)$ binders in an arbitrary order. In particular, $(\texttt{new}\ \emptyset\ \texttt{in}\ t)$ is just t.

6.4 Relating Target Terms back to Source Terms

Simple rules

$$\frac{}{x \sim (\emptyset, x)} \qquad \frac{}{\lambda x.\, t \sim (\emptyset, \lambda x.\, [\![t]\!])} \qquad \frac{t \sim (H_t, t') \qquad u \sim (H_u, u')}{t\ u \sim (H_t \uplus H_u, t'\ u')}$$

$$\frac{\left(t_i \sim (H_i, t_i')\right)^{i \in I}}{K\ (t_i)^{i \in I} \sim ((H_i)^{i \in I}, K\ \left(t_i'\right)^{i \in I})} \qquad \frac{t \sim (H, t')}{\texttt{match}\ t\ \texttt{with}\ h \sim (H, \texttt{match}\ t'\ \texttt{with}\ [\![h]\!])}$$

`letrec` rules

INIT

$$\frac{J \neq \emptyset}{\begin{array}{c}\texttt{let rec}\ \left(x_i = t_i\right)^{i \in I}, \left(x_j = t_j\right)^{j \in J}\ \texttt{in}\ u \\ \sim ([x_i \mapsto \bot]^{i \in I}, \texttt{new}\ \left(x_j\right)^{j \in J}\ \texttt{in}\ \texttt{par}(x_k \leftarrow t_k)^{k \in I \uplus J}; [\![u]\!])\end{array}}$$

WRITE

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i \in I} \qquad \left(t_{s,j} \sim (H_j, t_{t,j})\right)^{j \in J}}{\begin{array}{c}\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i \in I}, \left(y_j = t_{s,j}\right)^{j \in J}\ \texttt{in}\ u \\ \sim ([x_i \mapsto v_{t,i}]^{i \in I}[y_j \mapsto \bot]^{j \in J} \uplus (B_i)^{i \in I} \uplus \left(H_j\right)^{j \in J}, \texttt{par}((\texttt{Done})^{i \in I}, \left(y_j \leftarrow t_{t,j}\right)^{j \in J}); [\![u]\!])\end{array}}$$

DONE

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i \in I}}{\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i \in I}\ \texttt{in}\ u \sim ([x_i \mapsto v_{t,i}]^{i \in I} \uplus (B_i)^{i \in I} H, \texttt{Done}; [\![u]\!])}$$

FURTHER

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i \in I} \qquad u_s \sim (H, u_t)}{\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i \in I}\ \texttt{in}\ u_s \sim ([x_i \mapsto v_{t,i}]^{i \in I} \uplus (B_i)^{i \in I} \uplus H, u_t)}$$

Heap rules

HEAP-WEAKEN

$$\frac{t_s \sim (H, t_t) \qquad \forall x \in \mathrm{dom}(B),\ x \notin (H, t_t)}{t_s \sim (H \uplus B, t_t)}$$

HEAP-COPY

$$\frac{t_s \sim (H, t_t) \qquad \begin{array}{c}\phi \text{ compatible with } H \\ \mathrm{dom}(\phi) \subseteq \mathrm{dom}(H)\end{array}}{t_s \sim (\phi(H), \phi(t_t))}$$

Fig. 8. Head term relation $t \sim (H, t')$

We want to prove that this translation scheme is safe; that source terms that do not go vicious translate into target terms that do not segfault. This requires a backward simulation property: any reduction path from a translation in the target (in particular, a reduction path that leads to a segfault) needs to be simulated by a reduction path on the original source term (in particular, a reduction path that goes vicious).

We define a *head term relation* $t \sim (H, t')$ that relates the parts of a term and a configuration that are in reducible position. For non-reducible positions, we use the direct embedding $[\![t]\!]$ above. In

this relation, H contains not all the locations that are bound in t', but only those that correspond to `letrec`-binding found in t; intuitively, H is the disjoint union of all the `letrec`-bindings in t, seen as local store fragments.

The rules for the `letrec` constructs correspond to a decomposition of the various intermediate states of the reduction of their backpatching compilation.

Finally, the "heap rules" give reasoning principles to bridge the difference between the way the local and global stores evolve during reduction. In our source-level semantics, explicit substitutions (local store) may be duplicated or erased during reductions involving the values they belong to. Store duplication is expressed by a variable-renaming substitution ϕ, that "merges" different fragments of the global store together; the side-condition (ϕ compatible with H) guarantees that the resulting store is well-formed.

For reasons of space, we moved the explanation of this relation, including the definition of (ϕ compatible with H), as well as the proofs of its properties, to Appendix C (Compilation to Global Store: Simulation Proof) in the extended version. The two key results are mentioned here, guaranteeing that our analysis is also sound for this global-store semantics.

Theorem 4 (Backward Simulation).
If

$$t_s \sim (H, t_t) \qquad\qquad (H, t_t) \to (H', t'_t)$$

then $\exists t'_s$,

$$t_s \to^? t'_s \qquad\qquad t'_s \sim (H', t'_t)$$

Theorem 5 (Return-typed programs cannot segfault).

$$\emptyset \vdash t_s : \mathsf{Return} \quad \wedge \quad (\emptyset, [\![t_s]\!]) \to^* (H, t'_t) \quad \Longrightarrow \quad (H, t'_t) \notin \mathsf{Segfault}$$

Theorem 5 (Return-typed programs cannot segfault) guarantees that our analysis is sound for both our source language and its backpatching translation. Theorem 4 (Backward Simulation) also tells us that our source semantics has "enough" reduction rules compared to the global-store semantics. For example, if the global-store semantics computes a value for a term, then the source semantics would have computed a related value.

7 EXTENSION TO A FULL LANGUAGE

We now discuss the extension of our typing rules to the full OCaml language, whose additional features (e.g. exceptions, first-class modules and GADTs) contain subtleties that need special care.

7.1 The Size Discipline

The OCaml compilation scheme, one of several possible ways of treating recursive declarations, proceeds by reserving heap blocks for the recursively-defined values, and using the addresses of these heap blocks (which will eventually contain the values) as dummy values: it adds the addresses to the environment and computes the values accordingly. If no vicious term exists, the addresses are never dereferenced during evaluation, and evaluation produces "correct" values. Those correct values are then moved into the space occupied by the dummies, so that the original addresses contain the correct result.

This strategy depends on knowing how much space to allocate for each value. Not all OCaml types have a uniform size; e.g. variant types may contain constructors with different arities, resulting in different in-memory sizes, and the size of a closure depends on the number of free variables.

After checking that mutually-recursive definitions are meaningful using the rules we described, the OCaml compiler checks that it can realize them, by trying to infer a static size for each value. It then accepts to compile each declaration if either:

- it has a static size, or
- it doesn't have a statically-known size, but its usage mode of mutually-recursive definitions is always Ignore

(The second category corresponds to detecting some values that are actually non-recursive and lifting them out. Such non-recursive values often occur in standard programming practice, when it is more consistent to declare a whole block as a single **`let rec`** but only some elements are recursive.)

This static-size test may depend on lower-level aspects of compilation, or at least value representation choices. For example,

```
if p then (fun x -> x) else (fun x -> not x)
```

has a static size (both branches have the same size), but

```
if p then (fun x -> x + 1) else (fun x -> x + offset)
```

does not: the second function depends on a free variable `offset`, so it will be allocated in a closure with an extra field. (While `not` is also a free variable, it is statically resolvable to a global name.)

Relation to the mode system. The mode system corresponds to a correctness criterion on the operational semantics of programs; it is independent of compilation schemes. In contrast, the size discipline corresponds to a restrictive compilation strategy for value recursion that involves rejecting certain definitions. The size discipline is formalized by Hirschowitz et al. [2009]; it would be possible to incorporate it into our system, modelling it as a separate judgment to be checked for well-moded definitions (rather than as an enrichment of the mode judgment). However, the resulting system would be less portable to programming languages whose value representations differ from OCaml's, and which consequently would not use the same size discipline.

7.2 Dynamic Representation Checks: Float Arrays

OCaml uses a dynamic representation check for its polymorphic arrays: when the initial array elements supplied at array-creation time are floating-point numbers, OCaml chooses a specialized, unboxed representation for the array.

Inspecting the representation of elements during array creation means that although array construction looks like a guarding context, it is often in fact a dereference. There are three cases to consider: first, where the element type is statically known to be `float`, array elements will be unboxed during creation, which involves a dereference; second, where the element type is statically known not to be `float`, the inspection is elided; third, when the element type is not statically known the elements will be dynamically inspected — again a dereference.

The following program must be rejected, for example:

```
let rec x = (let u = [|y|] in 10.5)
    and y = 1.5
```

since creating the array `[|y|]` will unbox the element `y`, leading to undefined behavior if `y` — part of the same recursive declaration — is not yet initialized.

7.3 Exceptions and First-Class Modules

In OCaml, exception declarations are generative: if a functor body contains an exception declaration then invoking the functor twice will declare two exceptions with incompatible representations, so that catching one of them will not interact with raising the other.

Exception generativity is implemented by allocating a memory cell at functor-evaluation time (in the representation of the resulting module); and including the address of this memory cell as

an argument of the exception payload. In particular, creating an exception value `M.Exit 42` may dereference the module `M` where `Exit` is declared.

Combined with another OCaml feature, first-class modules, this generativity can lead to surprising incorrect recursive declarations, by declaring a module with an exception and using the exception in the same recursive block.

For instance, the following program is unsound and rejected by our analysis:

```
module type T = sig exception A of int end
let rec x = (let module M = (val m) in M.A 42)
and (m : (module T)) = (module (struct exception A of int end) : T)
```

In this program, the allocation of the exception value `M.A 42` dereferences the memory cell generated for this exception in the module `M`; but the module `M` is itself defined as the first-class module value `(m : (module T))`, part of the same recursive nest, so it may be undefined at this point.

(This issue was first pointed out by Stephen Dolan.)

7.4 GADTs

The original syntactic criterion for OCaml was implemented not directly on surface syntax, but on an intermediate representation quite late in the compiler pipeline (after typing, type-erasure, and some desugaring and simplifications). In particular, at the point where the check took place, exhaustive single-clause matches such as `match t with x -> ...` or `match t with () -> ...`) had been transformed into direct substitutions.

This design choice led to programs of the following form being accepted:

```
type t = Foo
let rec x = (match x with Foo -> Foo)
```

While this appears innocuous, it becomes unsound with the addition of GADTs to the language:

```
type (_, _) eq = Refl : ('a, 'a) eq
let universal_cast (type a) (type b) : (a, b) eq =
  let rec (p : (a, b) eq) = match p with Refl -> Refl in p
```

For the GADT `eq`, matching against `Refl` is not a no-op: it brings a type equality into scope that expands the set of types that can be assigned to the program [Garrigue and Rémy 2013]. It is therefore necessary to treat matches involving GADTs as inspections to ensure that a value of the appropriate type is actually available; without that change definitions such as `universal_cast` violate type safety.

7.5 Laziness

OCaml's evaluation is eager by default, but it supports an explicit form of lazy evaluation: the programmer can write `lazy e` and `force e` to delay and force the evaluation of an expression.

The OCaml implementation performs a number of optimizations involving **`lazy`**. For example, when the argument of **`lazy`** is a trivial syntactic value (variable or constant) for which eager and lazy evaluation behave equivalently, the compiler eliminates the allocation of the lazy thunk.

However, for recursive definitions eager and lazy evaluation are not equivalent, and so the recursion check must treat `lazy trivialvalue` as if the **`lazy`** were not there. For example, the following recursive definition is disallowed, since the optimization described above nullifies the delaying effect of the **`lazy`**

```
let rec x = lazy y and y = ...
```

while the following definition is allowed by the check, since the argument to **`lazy`** is not sufficiently trivial to be subject to the optimization:

```
let rec x = lazy (y+0) and y = ...
```

Our typing rule for **lazy** takes this into account: "trivial" thunks are checked in mode Return rather than Delay.

8 RELATED WORK

Degrees. Boudol [2001] introduces the notion of "degree" $\alpha \in \{0, 1\}$ to statically analyze recursion in object-oriented programs (recursive objects, lambda-terms). Degrees refine a standard ML-style type system for programs, with a judgment of the form $\Gamma \vdash t : \tau$ where τ is a type and Γ gives both a type and a degree for each variable. A context variable has degree 0 if it is required to evaluate the term (related to our Dereference), and 1 if it is not required (related to our Delay). Finally, function types are refined with a degree on their argument: a function of type $\tau^0 \to \tau'$ accesses its argument to return a result, while a $\tau^1 \to \tau'$ function does not use its argument right away, for example a curried function $\lambda x.\, \lambda y.\, (x, y)$ — whose argument is used under a delay in its body $\lambda y.\, (x, y)$. Boudol uses this reasoning to accept a definition such as `let rec obj = class_constructor obj params`, arising from object-oriented encodings, where `class_constructor` has a type $\tau^0 \to \ldots$.

Our system of mode is finer-grained than the binary degrees of Boudol; in particular, we need to distinguish Dereference and Guard to allow cyclic data structure constructions.

On the other hand, we do not reason about the use of function arguments at all, so our system is much more coarse-grained in this respect. In fact, refining our system to accept `let rec obj = constr obj params` would be incorrect for our use-case in the OCaml compiler, whose compilation scheme forbids passing yet-uninitialized data to a function.

In a general design aiming for maximal expressiveness, access modes should refine ML types; in Boudol's system, degrees are interlinked with the type structure in function types $\tau^\alpha \to \tau'$, but one could also consider pair types of the form ${\tau_1}^{\alpha_1} \times {\tau_2}^{\alpha_2}$, etc. In our simpler system, there are no interaction between value shapes (types) and access modes, so we can forget about types completely, a nice conceptual simplification. Our formalization is done entirely in an untyped fragment of ML.

Compilation. Hirschowitz, Leroy, and Wells [2003, 2009] discuss the space of compilation schemes for recursive value definitions. Their work is an inspiration for our own compilation result: they provide a source-level semantics based on floating bindings upwards in the term (similar to explicit substitutions or local thunk stores), and prove correctness of compilation to a global store with backpatching.

Our source-level semantics is close to theirs in spirit (we would argue that the use of *reduction at a distance* is an improvement), and our compilation scheme and its correctness proof are not novel compared to their work — they are there to provide additional intuition. The main contribution of our work is our mode system for recursive declarations, which is expressive enough to capture OCaml value definitions, yet simple and easy to infer.

A natural question for our work is whether the access-mode derivations we build in our safety check can inform the compilation strategy for recursive values. We concentrate on safety, but there is ongoing work by others on the compilation method for recursive values, which partly goes in this direction.

Fixing Letrec (Reloaded). Fixing Letrec (Reloaded) [Ghuloum and Dybvig 2009; Waddell, Sarkar, and Dybving 2005] is a nice brand of work from the Scheme community, centered on producing efficient code for recursive value declarations, even in presence of dynamic checks for the absence of uninitialized-name reads. It presents a static analysis, both for optimization purposes (eliding dynamic safety checks) and user convenience. The analysis is described by informal prose, but it is

similar in spirit to our mode analysis (using modes named "protected", which corresponds to our Delay, "protectable" which sounds like Return and "unsafe" which is Dereference). We precisely describe an analysis (richer, as it also has a Guard mode) and prove its correctness with respect to a dynamic semantics.

Name access as an effect. Dreyer [2004] proposes to track usage of recursively-defined variables as an effect, and designs a type-and-effect system whose effects annotations are sets of abstract names, maintained in one-to-one correspondence with **`let rec`**-bound variables. The construction **`let rec`** `X▷x : τ = e` introduces the abstract type-level name `X` corresponding to the recursive variable `x`. This recursive variable is made available in the scope of the right-hand-side `e : τ` at the type `box(X,τ)` instead of τ (reminding us of guardedness modalities). Any dereference of `x` must explicitly "unbox" it, adding the name `X` to the ambient effect.

This system is very powerful, but we view it as a core language rather than a surface language: encoding a specific usage pattern may require changing the types of the components involved, to introduce explicit `box` modalities:

- When one defines a new function from τ to τ', one needs to think about whether it may be later used with still-undefined recursive names as argument — assuming it indeed makes delayed uses of its argument. In that case, one should use the usage-polymorphic type function type $\forall X.\mathsf{box}(X, \tau) \to \tau'$ instead of the simple function type $\tau \to \tau'$. (It is possible to inject τ into $\mathsf{box}(X, \tau)$, so this does not restrict non-recursive callers.)
- One could represent cyclic data such as **`let rec`** `ones = 1 :: ones` in this system, but it would require a non-modular change of the type of the list-cell constructor from $\forall\alpha.\alpha \to \mathsf{List}(\alpha) \to \mathsf{List}(\alpha)$ to the box-expecting type $\forall\alpha.\alpha \to \forall X.\mathsf{box}(X, \mathsf{List}(\alpha)) \to \mathsf{List}(\alpha)$.

In particular, one cannot directly use typability in this system as a static analysis for a source language; this work needs to be complemented by a static analysis such as ours, or the safety has to be proved manually by the user placing box annotations and operations. However, we believe that any well-typed program that is accepted by our mode system could be encoded in Dreyer's system, roughly as follows:

- if in the context Γ of a derivation $\Gamma \vdash t : \mathsf{Return}$ in our system, we have $x : \mathsf{Dereference}$, then in the encoding the corresponding effect variable X would be an ambient capability ($\Gamma \vdash t : \tau[T]$ with $X \in T$)
- on the other hand, if we have $x : \mathsf{Return}$ or a more permissive mode, then we would give the corresponding term variable in the encoding type $\mathsf{box}(X, T)$

So, for example, $x : \mathsf{Dereference} \vdash x + 1 : \mathsf{Return}$ would be encoded as $X, x \vdash x + 1 : \mathrm{Int}[X]$ but $x : \mathsf{Guard} \vdash \{t = x\} : \mathsf{Return}$ would be encoded as $X, x : \mathsf{box}(X, \tau) \vdash \{t = x\} : \{t : \tau\}[\emptyset]$.

The whole derivation of an encoding of a valid recursive definition would have a non-boxed type on the right-hand side, without any of the effect variables of the recursively-defined variable in the ambient context.

Strictness analysis. Our analysis can be interpreted as a form of strictness or demand analysis, with modes above Return being non-strict and Dereference being the forcing mode. Note however that we use a may-analysis (a Dereference variable *may* be dereferenced) while strictness optimizations usually rely on a must-analysis (we only give Forcing when we know for sure that forcing happens); to do this one should change our interpretation of unknown functions to be conservative in the other direction, with mode Ignore rather than Dereference for their arguments. More importantly, strictness analyses typically try to compute more information than our modes: besides the question of whether a given subterm will be forced or not, they keep track of which prefixes of the possible

term shapes will get forced — this is more related to the finer-grained spaces of "ranks" or "degrees" for recursive functors.

Graph typing. Hirschowitz also collaborated on static analyses for recursive definitions in Bardou [2005]; Hirschowitz and Lenglet [2005]. The design goal was a simpler system than existing work aiming for expressiveness, with inference as simple as possible.

As a generalization of Boudol's binary degrees they use compactified numbers $\mathbb{N} \cup \{-\infty, \infty\}$. The degree of a free variable "counts" the number of subsequent λ-abstractions that have to be traversed before the variable is used; x has degree 2 in $\lambda y.\, \lambda z.\, x$. A $-\infty$ is never safe, it corresponds to our Dereference mode. 0 conflates our Guard and Return mode (an ad-hoc syntactic restriction on right-hand-sides is used to prevent under-determined definitions), the $n + 1$ are fine-grained representations of our Delay mode, and finally $+\infty$ is our Ignore mode.

Another salient aspect of their system is the use of "graphs" in the typing judgment: a use of `y` within a definition `let x = e` is represented as an edge from `y` to `x` (labeled by the usage degree), in a constraint graph accumulated in the typing judgment. The correctness criterion is formulated in terms of the transitive closure of the graph: if `x` is later used somewhere, its usage implies that `y` also needs to be initialized in this context.

One contribution of our work is to show that a more standard syntactic approach can replace the graph representation. Note that our typing rule for mutual-recursion uses a fixpoint computation, reminiscent of their transitive-closure computation but within a familiar type-system presentation.

Finally, their static analysis mentions the in-memory size of values, which needs to be known statically, in the OCaml compilation scheme, to create uninitialized memory blocks for the recursive names before evaluating the recursive definitions. Our mode system does not mention size at all, it is complemented by an independent (and simpler) analysis of static-size deduction, which is outside the scope of the present formalization, but described briefly in Section 7.1 (The Size Discipline).

F♯. Syme [2006] proposes a simple translation of mutually-recursive definitions into delay and force constructions that introduce and eliminate lazy values. For example, **let rec** $x = t$ **and** $y = u$ is turned into the following:

let rec x_{thunk} = **lazy** $(t[\text{force } x_{\text{thunk}}/x, \text{force } y_{\text{thunk}}/y])$
    and y_{thunk} = **lazy** $(u[\text{force } x_{\text{thunk}}/x, \text{force } y_{\text{thunk}}/y])$
let x = force x_{thunk} **and** y = force y_{thunk}

With this semantics, evaluation happens on-demand, which the recursive definitions evaluated at the time where they are first accessed. This implementation is very simple, but it turns vicious definitions into dynamic failures — handled by the lazy runtime which safely raises an exception. However, this elaboration cannot support cyclic data structures: The translation of `let rec ones = 1 :: ones` fails at runtime:

let rec $\text{ones}_{\text{thunk}}$ = **lazy** (1 :: force $\text{ones}_{\text{thunk}}$)

Furthermore, the translation affects the semantics of programs in surprising ways: in particular, the implicit introduction of laziness into definitions that start new threads can lead to unexpected multiple execution of computations and to race conditions.

Nowadays, F♯ provides an ad-hoc syntactic criterion, the "Recursive Safety Analysis" [Syme 2012], roughly similar to the previous OCaml syntactic criterion, that distinguishes "safe" and "unsafe" bindings in a mutually-recursive group; only the latter are subjected to the thunk-introducing translation.

Finally, the implementation also performs a static analysis to detect some definitions that are bound to fail — it over-approximates safety by ignoring occurrences within delaying terms (function abstractions or objects or lazy thunks) even if those delaying terms may themselves be used

(i.e. respectively called or accessed or forced) at definition time. We believe that we could recover a similar analysis by changing our typing rules for our constructions — but with the OCaml compilation scheme we must absolutely remain sound.

Needed computations. Further connections between laziness and recursive call-by-value definitions may be drawn: for example, Chang and Felleisen [2012] characterize call-by-need by introducing *needed computations*, which are similar in spirit to our idea of *forcing contexts*. However, the set of computations characterized by the two ideas are different in practice: for example, in our system `f (fst -)` is a forcing context, but the hole is not in "needed position" for call-by-need.

Intuitively, needed computations correspond to positions at which *any* choice of reduction order *must force* the computation to make progress, while forcing contexts correspond to positions where *some* choice of reduction order *may* force the computation (and fail if the value is initialized).

Operational semantics. Felleisen and Hieb [1992] and Ariola and Felleisen [1997] propose local-store semantics (for a call-by-value store and a call-by-need thunk store, respectively) that can express recursive bindings. The source-level operational semantics of Hirschowitz, Leroy, and Wells [2003, 2009] is more tailored to recursive bindings, manipulated as explicit substitutions, although the relation to standard explicit-substitution calculi is not made explicitly. They also provide a global-store semantics for their compilation-target language with mutable stores. Boudol and Zimmer [2002] and Dreyer [2004] use an abstract machine. Syme [2006] translates recursive definitions into lazy constructions, so the usual thunk-store semantics of laziness can be used to interpret recursive definitions. Finally, Nordlander, Carlsson, and Gill [2008] give the simplest presentation of a source-level semantics we know of; we extend it with algebraic datatypes and pattern-matching, and use it as a reference to prove the soundness of our analysis.

Our own experience presenting this work is that local-store semantics has been largely forgotten by the programming-language community, which is a shame as it provides a better treatment of recursive definitions (or call-by-need) than global-store semantics.

One inessential detail in which the semantics often differ is the evaluation order of mutually-recursive right-hand-sides. Many presentations enforce an arbitrary (e.g. left-to-right) evaluation order. Some systems [Nordlander, Carlsson, and Gill 2008; Syme 2006] allow a reduction to block on a variable whose definition is not yet evaluated, and go evaluate it in turn; this provides the "best possible order" for the user. Another interesting variant would be to say that the reduction order is unspecified, and that an uninitialized variable is a stuck term whose evaluation results in a fatal error; this provides the "worst possible order", failing as much as possible; as far as we know, the previous work did not propose it, although it is a simple presentation change. Most static analyses are evaluation-order-independent, so they are sound and complete with respect to the "worst order" interpretation.

9 CONCLUSION

We have presented a new static analysis for recursive value declarations, designed to solve a fragility issue in the OCaml language semantics and implementation. It is less expressive than previous works that analyze function calls in a fine-grained way; in return, it remains fairly simple, despite its ability to scale to a fully-fledged programming language, and the constraint of having a direct correspondence with a simple inference algorithm.

We believe that this static analysis may be of use for other functional programming languages, both typed and untyped. It also seems likely that the techniques we have used in this work will apply to other systems — type parameter variance, type constructor roles, and so on. Our hope in carefully describing our system is that we will eventually see a pattern emerge for the design and structure of "things that look like type systems" in this way.

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. In *ICFP*.

Beniamino Accattoli. 2013. Evaluating functions as processes. In *TERMGRAPH*.

Beniamino Accattoli and Delia Kesner. 2010. The structural lambda-calculus. (Oct. 2010). working paper or preprint.

Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need lambda Calculus. *J. Funct. Program.* 7, 3 (1997), 265–301.

Romain Bardou. 2005. *Typage des modules récursifs en Caml.* Technical Report. ENS Lyon.

Gérard Boudol. 2001. The Recursive Record Semantics of Objects Revisited. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 269–283.

Gérard Boudol and Pascal Zimmer. 2002. Recursion in the call-by-value lambda-calculus. In *FICS*. 61–66.

Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *PACMPL* 2, ICFP (2018).

Stephen Chang and Matthias Felleisen. 2012. The Call-by-Need Lambda Calculus, Revisited. In *ESOP (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 128–147.

Derek Dreyer. 2004. A Type System for Well-founded Recursion. In *POPL*. ACM, New York, NY, USA, 293–305.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271.

Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *APLAS*. 257–272.

Samir Genaim and Michael Codish. 2001. Inferring termination conditions for logic programs using backwards analysis. In *APPIA-GULP-PRODE 2001: Joint Conference on Declarative Programming, Évora, Portgual, September 26-28, 2001, Proceedings, Évora, Portugal, September 26-28, 2001*, Luís Moniz Pereira and Paulo Quaresma (Eds.). Departamento de Informática, Universidade de Évora, 229–243.

Abdulaziz Ghuloum and R. Kent Dybvig. 2009. Fixing Letrec (reloaded). In *Scheme Workshop*.

Tom Hirschowitz and Sergueï Lenglet. 2005. *A practical type system for generalized recursion.* Technical Report. ENS Lyon.

Tom Hirschowitz, Xavier Leroy, and J. B. Wells. 2003. Compilation of Extended Recursion in Call-by-value Functional Languages. In *PPDP*. ACM, New York, NY, USA, 160–171.

Tom Hirschowitz, Xavier Leroy, and J. B. Wells. 2009. Compilation of Extended Recursion in Call-by-value Functional Languages. *Higher Order Symbol. Comput.* 22, 1 (March 2009).

John Hughes. 1987. Backwards Analysis of Functional Programs. In *Partial Evaluation and Mixed Computation*.

Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae* 150 (2017), 347–377.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *FLOPS*. 86–102.

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154.

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

Johan Nordlander, Magnus Carlsson, and Andy J. Gill. 2008. Unrestricted Pure Call-by-value Recursion. In *ML Workshop*.

Ilya Sergey, Simon Peyton-Jones, and Dimitrios Vytiniotis. 2017a. Theory and Practice of Demand Analysis in Haskell. draft.

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017b. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.* 27 (2017), e11.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301.

Don Syme. 2005. *An Alternative Approach to Initializing Mutually Referential Objects.* Technical Report MSR-TR-2005-31. 27 pages.

Don Syme. 2006. Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge. *Electron. Notes Theor. Comput. Sci.* 148, 2 (2006), 3–25.

Don Syme. 2012. The Fsharp language reference, Versions 2.0 to 4.1, Section 14.6.6, Recursive Safety Analysis.

Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybving. 2005. Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct. *Journal of Higher-Order and Symbolic Computation* 18 (2005).

A PROPERTIES OF OUR TYPING JUDGMENT

The following technical results can be established by simple inductions on typing derivations, without any reference to an operational semantics.

Lemma 2 (Ignore inversion). $\Gamma \vdash t : \mathsf{Ignore}$ *is provable with only* Ignore *in* Γ.

Lemma 3 (Delay inversion). $\Gamma \vdash t : \mathsf{Delay}$ *holds exactly when* Γ *maps all free variables of* t *to* Delay *or* Ignore.

Lemma 4 (Dereference inversion). $\Gamma \vdash t : \mathsf{Dereference}$ *holds exactly when* Γ *maps all free variables of* t *to* Dereference.

Lemma 5 (Environment flow). *If a derivation* $\Gamma \vdash t : m$ *contains a sub-derivation* $\Gamma' \vdash t' : m'$, *then* $\forall x \in \Gamma,\ \Gamma(x) \geq \Gamma'(x)$.

Lemma 6 (Weakening). *If* $\Gamma \vdash t : m$ *holds then* $\Gamma + \Gamma' \vdash t : m$ *also holds.*

(Weakening would not be admissible if our variable rule imposed Ignore on the rest of the context.)

Lemma 7 (Substitution). *If* $\Gamma, x : m_u \vdash t : m$ *and* $\Gamma' \vdash u : m_u$ *hold, then* $\Gamma + \Gamma' \vdash t[u/x] : m$ *holds.*

Lemma 8 (Subsumption elimination). *Any derivation in the system can be rewritten so that the subsumption rule is only applied with the variable rule as premise.*

Theorem (1: Principal environments). *Whenever both* $\Gamma_1 \vdash t : m$ *and* $\Gamma_2 \vdash t : m$ *hold, then* $\min(\Gamma_1, \Gamma_2) \vdash t : m$ *also holds.*

Proof. The proof first performs subsumption elimination on both derivations, and then by simultaneous induction on the results. The elimination phase makes proof syntax-directed, which guarantees that (on non-variables) the same rule is always used on both sides in each derivation. □

This results tells us that whenever $\Gamma \vdash t : m$ holds, then it holds for a minimal environment Γ — the minimum of all satisfying Γ.

Definition 2 (Minimal environment). Γ *is minimal for* $t : m$ *if* $\Gamma \vdash t : m$ *and, for any* $\Gamma' \vdash t : m$ *we have* $\Gamma \preceq \Gamma'$.

In fact, we can give a precise characterization of "minimal" derivations, that uniquely determines the output of our backwards analysis algorithm.

Definition 3 (Minimal binding rule). *An application of the binding rule is* minimal *exactly when the choice of* Γ'_i *is the least solution to the recursive equation in its third premise.*

Definition 4 (Minimal derivation). *A derivation is* minimal *if it does not use the subsumption rule, each binding rule is minimal and, in the conclusion* $\Gamma \vdash x : m$ *of each variable rule,* Γ *is minimal for* $x : m$.

Definition 5 (Minimization). *Given a derivation* $D :: \Gamma \vdash t : m$, *we define the (minimal) derivation* $\mathsf{minimal}(D)$ *by:*

- *Turning each binding rule into a minimal version of this binding rule — this may require applying Lemma 6 (Weakening) to the* `let rec` *derivation below.*
- *Performing subsumption-elimination to get another derivation of* $\Gamma \vdash t : m$.
- *Replacing the context of each variable rule by the minimal context for this variable — this does not introduce new subsumptions.*

Fact 1 (Minimality). *If* $D :: \Gamma \vdash t : m$ *and* $\mathsf{minimal}(D) :: \Gamma_m \vdash t : m$, *then* $\Gamma_m \preceq \Gamma$.

Lemma 9 (Stability). *If* D *is a minimal derivation, then* $\mathsf{minimal}(D) = D$.

Lemma 10 (Determinism). *If* $D_1 :: \Gamma_1 \vdash t : m$ *and* $D_2 :: \Gamma_2 \vdash t : m$, *then* $\mathsf{minimal}(D_1)$ *and* $\mathsf{minimal}(D_2)$ *are the same derivation.*

Corollary 2 (Minimality equivalence). *The environment* Γ *of a derivation* $\Gamma \vdash t : m$ *is minimal for* $t : m$ *if and only if* $\Gamma \vdash t : m$ *admits a minimal derivation.*

Proof. If Γ is minimal for $t : m$, then the context $\Gamma_m \preceq \Gamma$ obtained by minimization must itself be Γ.

Conversely, if a derivation $D_m :: \Gamma \vdash t : m$ is minimal, then all other derivations $\Gamma' \vdash t : m$ have D_m as minimal derivation by Lemma 9 (Stability) and Lemma 10 (Determinism), so $\Gamma \preceq \Gamma'$ holds. □

Theorem 6 (Localization). $\Gamma \vdash t : m'$ *implies* $m\,[\Gamma] \vdash t : m\,[m']$.
Furthermore, if Γ *is minimal for* $t : m'$, *then* $m\,[\Gamma]$ *is minimal for* $t : m\,[m']$.

Proof. The proof proceeds by direct induction on the derivation, and does not change its structure: each rule application in the source derivation becomes the same source derivation in the result. In particular, minimality of derivations is preserved, and thus, by Corollary 2 (Minimality equivalence), minimality of environments is preserved.

Besides associativity of mode composition, many cases rely on the fact that external mode composition preserves the mode order structure: $m'_1 \prec m'_2$ implies $m\left[m'_1\right] \prec m\left[m'_2\right]$, and $\max(m\left[m'_1\right], m\left[m'_2\right])$ is $m\left[\max(m'_1, m'_2)\right]$. □

B PROOFS FOR Section 5.3 (Soundness)

Lemma 11 (1: Forcing modes).
If $\Gamma, x : m_x \vdash E_\mathsf{f}[x] : m$ *with* $m \geq \mathsf{Return}$, *then also* $m_x \geq \mathsf{Return}$.

Proof. E_f may be of the form L or $E[F_\mathsf{f}[L]]$
In the case of binding contexts L we have $m_x = \mathsf{Return}$ by construction.
In the case with a forcing frame, $E_\mathsf{f} = E[F_\mathsf{f}[L]]$, let us call m_E the mode of the hole of E. It is immediate that the mode imposed by L on its hole is Return, and that the mode imposed by F_f on its own hole is Dereference, so the total mode m_x is $m_E\,[\mathsf{Dereference}\,[\mathsf{Return}]]$. We can prove by an easy induction on E that m_E is not Ignore or Delay – those are not evaluation contexts, so we have $m_E \geq \mathsf{Guard}$. We conclude by monotonicity of mode composition:

$$m_x = m_E\,[\mathsf{Dereference}\,[\mathsf{Return}]] \geq \mathsf{Guard}\,[\mathsf{Dereference}\,[\mathsf{Return}]] = \mathsf{Dereference}$$

□

Theorem 7 (2: Vicious). $\emptyset \vdash t : \mathsf{Return}$ *never holds for* $t \in \mathsf{Vicious}$.

Proof. Given $\vdash t : \mathsf{Return}$, let us assume that t is $E[x]$ with no value binding for x in E, and show that E is not a forcing context.

We implicitly assume that all terms are well-scoped, so the absence of value binding means that x occurs in a **`let rec`** binding still being evaluated somewhere in E: $E[x]$ is of the form

$$E[x] = E_\mathsf{out}[t_\mathsf{rec}] \qquad\qquad t_\mathsf{rec} = (\mathtt{let\ rec}\ b, y = E_\mathsf{in}[x], b'\ \mathtt{in}\ u)$$

where x is bound in b, b' or is y itself.

Given our **`let rec`** typing rule (see Figure 3), the typing derivation for t contains a sub-derivation for t_{rec} of the form

$$\frac{\left(\Gamma_i, \left(x_j : m_{i,j}\right)^j \vdash t_i : \text{Return}\right)^i \qquad \left(m_{i,j} \preceq \text{Guard}\right)^{i,j} \qquad \left(\Gamma_i' = \Gamma_i + \sum \left(m_{i,j}\left[\Gamma_j'\right]\right)^j\right)^i}{\left(x_i : \Gamma_i'\right)^i \vdash \texttt{rec}\ \left(x_i = t_i\right)^i}$$

In particular, the premise for $E_{\text{in}}[x]$ is of the form $\Gamma, \left(x_j : m_j\right)^j \vdash E_{\text{in}}[x] : \text{Return}$ with $\left(x_j \preceq \text{Guard}\right)^j$, and in particular $x \preceq \text{Guard}$ so $x \not\preceq \text{Return}$.

By Lemma 1 (Forcing modes), E_{in} cannot be a forcing context, and in consequence E is not forcing either. □

Theorem 8 (3: Subject reduction). *If* $\Gamma \vdash t : m$ *and* $t \to t'$ *then* $\Gamma \vdash t' : m$.

Proof. We reason by inversion on the typing derivation of redexes, first for head-reduction $t \to^{\text{hd}} t'$ and then for reduction $t \to t'$.

Head reduction. We only show the head-reduction case for functions; pattern-matching is very similar. We have:

$$\dfrac{\dfrac{\dfrac{\Gamma_t, x : m_x \vdash t : m\,[\text{Dereference}]\,[\text{Delay}]}{\Gamma_t \vdash \lambda x.\, t : m\,[\text{Dereference}]}}{\Gamma_t \vdash L[\lambda x.\, t] : m\,[\text{Dereference}]} \qquad \Gamma_v \vdash v : m\,[\text{Dereference}]}{\Gamma_t + \Gamma_v \vdash L[\lambda x.\, t]\ v : m}$$

By associativity, $m\,[\text{Dereference}]\,[\text{Delay}]$ is the same as $m\,[\text{Dereference}]$.

By subsumption, $\Gamma_t, x : m_x \vdash t : m\,[\text{Dereference}]$ implies $\Gamma_t, x : m_x \vdash t : m$.

To conclude by using Lemma 7 (Substitution), we must reconcile the mode of the argument $v : m\,[\text{Dereference}]$ with the (apparently arbitrary) mode $x : m_x$ of the variable. We reason by an inelegant case distinction.

- If $m\,[\text{Dereference}]$ is Dereference, then by inversion (Lemma 4) either m_x is Dereference (problem solved) or x does not occur in t (no need for the substitution lemma).
- If $m\,[\text{Dereference}]$ is not Dereference, then m must be Ignore or Delay. If it is Ignore, inversion (Lemma 2) directly proves our goal. If it is Delay, then by inversion (Lemma 3) m_x itself can be weakened (subsuming the derivation of t) to be below Delay.

Reduction under context. Reducing a head-redex under context preserves typability by the argument above. Let us consider the lookup case.

$$\frac{(x = v) \overset{\text{ctx}}{\in} E}{E[x] \to E[v]}$$

By inspecting the $(x = v) \overset{\text{ctx}}{\in} E$ derivation, we find a value binding B within E with $x = v$, and a derivation of the form

$$\frac{\left(x_i : \Gamma'_i\right)^i \vdash \texttt{rec}\ B \qquad \left(m'_i\right)^i \overset{\text{def}}{=} \left(\max(m_i, \text{Guard})\right)^i \qquad \Gamma_u, \left(x_i : m_i\right)^i \vdash u : m}{\sum \left(m'_i \left[\Gamma'_i\right]\right)^i + \Gamma_u \vdash \texttt{let rec}\ B\ \texttt{in}\ u : m}$$

$$\frac{\left(\Gamma_i, \left(x_j : m_{i,j}\right)^j \vdash v_i : \text{Return}\right)^i \qquad \left(m_{i,j} \preceq \text{Guard}\right)^{i,j} \qquad \left(\Gamma'_i = \Gamma_i + \sum \left(m_{i,j} \left[\Gamma'_j\right]\right)^j\right)^i}{\left(x_i : \Gamma_i\right)^i \vdash \texttt{rec}\ \left(x_i = v_i\right)^i}$$

By abuse of notation, we will write m_x, Γ_x and Γ'_x to express the m_i, Γ_i and Γ'_i for the i such that $x_i = x$.

The occurrence of x in the hole of $E[\Box]$ is typed (eventually by a variable rule) at some mode $m_\Box$. The declaration-side mode m_x was built by collecting the usage modes of all occurrences of x in the **`let rec`** body u, which in particular contains the hole of E, so we have $m_\Box \preceq m_x$ by Lemma 5 (Environment flow).

The binding derivation gives us a proof $\Gamma_x, \Gamma_{\texttt{rec}} \vdash v : \text{Return}$ that the binding $x = v$ was correct at its definition site, where $\Gamma_{\texttt{rec}}$ has exactly the mutually-recursive variables $\left(x_i : m_i\right)^i$. Notice that this subderivation is completely independent of the ambient expected mode m.

By Theorem 6 (Localization), we can compose this within $m_\Box$ to get a derivation $m_\Box \left[\Gamma_x, \Gamma_{\texttt{rec}}\right] \vdash v : m_\Box$, that we wish to substitute into the hole of E. First we weaken it (Lemma 6) into the judgment $m_x \left[\Gamma_x, \Gamma_{\texttt{rec}}\right] \vdash v : m_\Box$.

Plugging this derivation in the hole of E requires weakening the derivation of u (the part of $E[\Box]$ that is after the declaration of x) to add the environment $m_x \left[\Gamma_x, \Gamma_{\texttt{rec}}\right]$. Weakening is always possible (Lemma 6), but it may change the environment of the derivation, while we need to preserve the environment of $E[x]$. Consider the following valid derivation:

$$\frac{\left(x_i : \Gamma'_i\right)^i \vdash \texttt{rec}\ B \qquad \left(m''_i\right)^i \overset{\text{def}}{=} \left(\max(\max(m_i, m_x \left[\Gamma_{\texttt{rec}}\right] (x_i)), \text{Guard})\right)^i \qquad \Gamma_u + m_x \left[\Gamma_x\right], \left(x_i : m_i\right)^i + m_x \left[\Gamma_{\texttt{rec}}\right] \vdash u[v/x] : m}{\sum \left(m''_i \left[\Gamma'_i\right]\right)^i + \Gamma_u + m_x \left[\Gamma_x\right] \vdash \texttt{let rec}\ B\ \texttt{in}\ u[v/x] : m}$$

To show that we preserve the environment of $E[x]$, we show that this derivation is not in a bigger environment than the environment of our source term:

$$\sum \left(m''_i \left[\Gamma'_i\right]\right)^i + \Gamma_u + m_x \left[\Gamma_x\right] \qquad \preceq \qquad \sum \left(m'_i \left[\Gamma'_i\right]\right)^i + \Gamma_u$$

By construction we have $m_x \preceq m'_x \preceq m''_x$ and $\Gamma_x \preceq \Gamma'_x$, so $m_x \left[\Gamma_x\right] \preceq m''_x \left[\Gamma'_x\right]$ which implies

$$\sum \left(m''_i \left[\Gamma'_i\right]\right)^i + \Gamma_u + m_x \left[\Gamma_x\right] \qquad \preceq \qquad \sum \left(m''_i \left[\Gamma'_i\right]\right)^i + \Gamma_u$$

Then, notice that $\Gamma_{\texttt{rec}}(x_i)$ is exactly $m_{x,i}$, so m''_i is $\max(m'_i, m_x \left[m_{x,i}\right])$. We can thus rewrite $m''_i \left[\Gamma'_i\right]$ into $m'_i \left[\Gamma\right]'_i + m_x \left[m_{x_i}\right] \left[\Gamma'_i\right]$, which gives

$$\sum \left(m''_i \left[\Gamma'_i\right]\right)^i + \Gamma_u \qquad = \qquad \sum \left(m'_i \left[\Gamma'_i\right]\right)^i + m_x \left[\sum \left(m_{x,i} \left[\Gamma'_i\right]\right)^i\right] + \Gamma_u$$

The extra term $\sum \left(m_{x,i} \left[\Gamma'_i\right]\right)^i$ is precisely the term that appears in the definition of Γ'_x from the $\left(\Gamma_i\right)^i$, taking into account transitive mutual dependencies – indeed, when we replace x by its value

v, we replace transitive dependencies on its mutual variables by direct dependencies on occurrences in v. We thus have

$$\sum \left(m_{x,i} \left[\Gamma_i'\right]\right)^i \preceq \Gamma_x'$$

and can conclude with

$$\begin{array}{rl} & \sum \left(m_i' \left[\Gamma_i'\right]\right)^i + m_x \left[\sum \left(m_{x,i} \left[\Gamma_i'\right]\right)^i\right] + \Gamma_u \\ \preceq & \sum \left(m_i' \left[\Gamma_i'\right]\right)^i + m_x \left[\Gamma_x'\right] + \Gamma_u \\ \preceq & \sum \left(m_i' \left[\Gamma_i'\right]\right)^i + \Gamma_u \end{array}$$

□

C COMPILATION TO GLOBAL STORE: SIMULATION PROOF

This section explains the definition of the relation in Figure 8, and develops the simulation result to prove Theorem 4 (Backward Simulation) and Theorem 5 (Return-typed programs cannot segfault).

Remark 4 (α-equivalence of configurations). *In a configuration (H, t), the heap H binds the variables of its domain in t. In particular, consistently renaming a variable in H and in t results in an α-equivalent configuration; for example $([x \mapsto \bot], x)$ and $([y \mapsto \bot], y)$ are α-equivalent. Our operations on configurations respect α-equivalence.*

Fact 2 (Congruence). *If $(H, t) \to (H', t')$ then $(H_E \uplus H, E[t]) \to (H_E \uplus H', E[t'])$.*

Fact 3 (Relevant head reduction). *If $(H, t) \to^{\mathsf{hd}} (H', t')$ then the domain of H contains only free variables of t; furthermore, t' and the domain of H' contain only free or bound variables of t.*

C.1 Relating the Local-Store Term with Global-Store Configurations

C.1.1 Simple rules. The simple rules relate term-formers that exist in both the source and target language; they simply ask the evaluable subterms to be related and take the union of the corresponding heaps.

C.1.2 `letrec` *rules.* The `letrec` rules relate the `letrec` construct in the source language to store-update operations in the target. There are several rules, that correspond to distinct "moments" in the computation of the translation of a $\texttt{let rec } (x_i = t_i)^{i \in I} \texttt{ in } u$ binding, which initially gets translated as $\texttt{new } (x_i)^i \texttt{ in par}(x_i \leftarrow t_i)^i; [\![u]\!]$:

- In the first moment we are reducing the allocations $\texttt{new } (x_i)^i \texttt{ in}$: some of them have already been allocated in the heap, others are yet to be evaluated. The rest of the target term is not in reducible position, so it is unchanged. This is the rule init.
- In the second moment, all store locations have been created, and we are evaluating the store updates $(x_i \leftarrow t_i)^i$. Some of them have been fully reduced to values and then written in the store, others are still (partially evaluated) terms. This is the rule write.
- In the third moment, the store updates have been performed so the par ... block reduced to `Done`. This is the rule done.
- In the fourth and last moment, we are in the process of further evaluating the body of the `letrec`-definition, u; the target is not restricted to the source translation $[\![u]\!]$ anymore, it may be any target configuration related to u. This is the rule further.

Example 2. *The following relations hold:*

$$x\ (\texttt{let rec } y = \texttt{S } y \texttt{ in } y) \sim (\emptyset, x\ (\texttt{new } y \texttt{ in par}(y \leftarrow \texttt{S } y); y))$$

$$x\ (\texttt{let rec } y = \texttt{S } y \texttt{ in } y) \sim ([y \mapsto \bot], x\ (\texttt{par}(y \leftarrow \texttt{S } y); y))$$

$x\ (\texttt{let rec}\ y = \texttt{S}\ y\ \texttt{in}\ y) \sim ([y \mapsto \texttt{S}\ y], x\ y) \qquad (\texttt{let rec}\ y = \texttt{S}\ y\ \texttt{in}\ x\ y) \sim ([y \mapsto \texttt{S}\ y], x\ y)$

C.1.3 Heap rules. Finally, the heap rules give more to relate local heaps and global heaps (than simple disjoint union of all local heaps), to account for the fact that local heaps (explicit substitutions) may be duplicated or erased during the reduction of our source terms, as demonstrated in our Example 1.

When trying to prove that a source term is related to a target configuration, the HEAP-WEAKEN rule allows let us discard a fully-evaluated part of the global target store that is not referenced from the rest of the heap or the target term. This lets us ignore, when relating two terms, the parts of the global store that correspond to local stores that have been erased by β-reduction, for example when reducing the source term $(\lambda x.\, \texttt{Done})\ (\texttt{let rec}\ x = \texttt{Foo}\ \texttt{in}\ x)$ and its related target term.

REMARK 5. *The restriction that only value heaps B may be weakened, instead of arbitrary heap fragments H', gives us a stronger inversion principle in Fact 7: in certain cases we can prove that the heap of a related configuration must be a value heap, which would never be the case if one could always weaken arbitrary heaps.*

The HEAP-COPY rule let us relate a source term and a target configuration even when the source term contains several copies of a local store fragment that is included only once in the target global store: by this rule it suffices to prove the relation on a larger global store that includes several copies of some parts of the original store.

The rule is defined for any *renaming* ϕ from heap locations (variables) to locations, that may "collapse" several locations from the larger heap H into the same location in the resulting heap $\phi(H)$. To apply this rule we require that ϕ be compatible with the heap H, which intuitively means that locations x, y in H that are collapsed by ϕ into in the same location must have pointed to the same value after ϕ-renaming. We also require ϕ be the identity on variables outside $\mathrm{dom}(H)$; without this restriction, applying ϕ could change the free variables of the target term.

DEFINITION 6 (FINITE RENAMING). *A (finite)* renaming ϕ *is a total function from variables to variables whose domain* $\mathrm{dom}(\phi)$*, defined as the set* $\{x \mid \phi(x) \neq x\}$*, is finite.*

We write $\phi(t)$ *for the replacement of each free variable* $x \in t$ *by* $\phi(x)$*.*

DEFINITION 7 (HEAP COMPATIBILITY). *We write* (ϕ *compatible with* H) *if the following properties hold:*

functionality $\forall x, y \in \mathrm{dom}(H), \quad \phi(x) = \phi(y) \implies \phi(H(x)) = \phi(H(y))$
definedness $\forall x, y \in \mathrm{dom}(H), \quad \phi(x) = \phi(y) \wedge H(x) = \bot \implies x = y$

Functionality implies that $\phi(H) \stackrel{\text{def}}{=} \{\phi(x) \mapsto \phi(t) \mid (x \mapsto t) \in H\}$ is well-defined as a finite map.

Definedness implies that ϕ is injective on the uninitialized locations of H: distinct uninitialized locations cannot be collapsed together.

EXAMPLE 3. *The following relations hold:*

$$x\ (\texttt{let rec}\ y = \texttt{S}\ y\ \texttt{in}\ y)\ (\texttt{let rec}\ y' = \texttt{S}\ y'\ \texttt{in}\ y') \sim ([y \mapsto \texttt{S}\ y][y' \mapsto \texttt{S}\ y'], x\ y\ y')$$

$$x\ (\texttt{let rec}\ y = \texttt{S}\ y\ \texttt{in}\ y)\ (\texttt{let rec}\ y' = \texttt{S}\ y'\ \texttt{in}\ y') \sim ([y \mapsto \texttt{S}\ y], x\ y\ y)$$

The second relation is proved by using the HEAP-COPY rule on the first one, with the heap function ϕ defined by $\phi(y') = \phi(y) = y$ and, for any $z \neq y'$, $\phi(z) = z$. This ϕ is compatible with the example heap because $\phi(\texttt{S}\ y) = \phi(\texttt{S}\ y') = \texttt{S}\ y$.

Fact 4 (Union compatibility). *If ϕ is compatible with H_1 and H_2, and $\phi(H_1)$ and $\phi(H_2)$ have disjoint domains, then ϕ is compatible with $H_1 \uplus H_2$.*

Properties of the relation. The simple rules are syntax-directed. The others are not:

- The `letrec`-binding rules relate the same source term to several different target terms. In the case of the rule further, the target term is exactly the one of a premise: any term u_t may be related to a term of the form `let rec` B `in` ... if it is the result of a reduction sequence that introduced B in the global heap.
- The heap rules act only on the target heap, not on the source or target terms. In particular, they may occur at the conclusion of any relation derivation.

Fact 5. *A related term and configuration have the same free variables.*

Fact 6. *For any source term t we have $t \sim (\emptyset, [\![t]\!])$.*

Proof. By immediate induction on t. In the `letrec` case, use the init rule with $I = \emptyset$ — no location initialized yet. For all other term-formers, use the simple rules. □

Fact 7 (Target value inversion). *If we have $t \sim (H, v_t)$, a relation where the target term is a value, then:*

- *t is a source value of the form $L[v_s]$,*
- *H is a value heap B,*
- *$v_s \sim (\emptyset, v_t)$ are related by a simple rule in the derivation of $L[v_s] \sim (B, v_t)$.*

Proof. By induction on the relation derivation.

The property is directly satisfied by the simple rules that may related a target value (λ-abstraction and data constructor), with $L = \Box$ and $B = \emptyset$.

The only letrec-related rule whose target may be a value is further, and it preserves this property (growing L and B with the recursive bindings).

The heap rules also preserve this property; they may only transform the value heap B into another value heap. This is by definition for heap-weaken: it is restricted to weakening value heaps only. In the case of heap-copy, remark that if $\phi(H)$ is a value heap, then H must also be a value heap: ϕ is a variable renaming, so we can only have $\phi(v?) = \bot$ if $v? = \bot$. □

Note that we do not have such an inversion principle for source values: if we have $v_s \sim (H, t)$, then the target source term t need not be a value, as v_s may contain value bindings that are not yet evaluated in t. For example we have

$$\texttt{let rec } y = \mathsf{S}\, y \texttt{ in } \mathsf{S}\, y \sim ([y \mapsto \bot], \mathsf{par}(y \leftarrow \mathsf{S}\, y); \mathsf{S}\, y)$$

where the source term is a value.

C.2 Backward Simulation

Lemma 12 (Relation substitution).
If

$$t_s \sim (H, t_t) \qquad u_s \sim (B, u_t) \qquad H, B \text{ disjoint}$$

then

$$t_s[u_s/x] \sim (H \uplus B, t_t[u_t/x])$$

Proof. If x does not occur in t_t, then this is exactly the heap-weaken rule. Let us now consider the case where there is at least one occurrence of $x \in t_t$. Let $(x_i)^{i \in I}$ be the (non-empty) family of occurrences of x in t_t.

Let us choose, for each x_i, an α-equivalent copy $(B_i, u_{t,i})$ of (B, u_t), where B_i is fresh for (H, t_t). From $t_s \sim (H, t_t)$ we get a derivation of

$$t_s[u_s/x] \sim (H \biguplus_{i \in I} B_i, t_t[u_{t,i}/x_i]^{i \in I})$$

by replacing each sub-derivation $x \sim (\emptyset, x_i)$ by our assumptions $u_s \sim (B_i, u_{t,i})$.

We now consider the map ϕ that maps each $y_k \in B_k$ for some k to the corresponding $y \in B$, and all other variables to themselves. By this definition we have that:

- $\mathrm{dom}(\phi) \subseteq \biguplus_i B_i$;
- ϕ is the identity on H, so it is compatible with H;
- $\phi(B_i) = B$ and $\big(\phi(u_{t,i}) = u_t\big)^i$;
- ϕ is functional on $\biguplus_i B_i$; definedness trivially holds on value heaps, so ϕ is compatible with $\biguplus_i B_i$.

Furthermore, H and $\phi(\biguplus_i B_i = B$ have disjoint domains, so by Fact 4 we have that ϕ is compatible with $H \biguplus_i B_i$.

As a consequence, from

$$t_s[u_s/x] \sim (H \biguplus_{i \in I} B_i, t_t[u_{t,i}/x_i]^{i \in I})$$

we can apply the HEAP-COPY rule with ϕ to deduce

$$t_s[u_s/x] \sim (\phi(H \biguplus_{i \in I} B_i), \phi(t_t[u_{t,i}/x_i]^{i \in I}))$$

that is our goal

$$t_s[u_s/x] \sim (H \uplus B, t_t[u_t/x])$$

□

THEOREM (4: BACKWARD SIMULATION).
If

$$t_s \sim (H, t_t) \qquad (H, t_t) \to (H', t'_t)$$

then $\exists t'_s$,

$$t_s \to^? t'_s \qquad t'_s \sim (H', t'_t)$$

PROOF. We proceed by induction on the derivation of $t_s \sim (H, t_t)$.

The "simple rules" are the easy cases, with the interesting cases coming heap-management or letrec-related cases.

Simple rules. The cases of variables and λ-abstractions are immediate, as no reduction can occur. For application, matching and constructors, the same reasoning works in all cases, so we will only consider application.

$$\frac{f_s \sim (H_f, f_t) \qquad u_s \sim (H_u, u_t)}{f_s\ u_s \sim (H_f \uplus H_u, f_t\ u_t)}$$

We proceed by case analysis on the reduction $(H, t_t) \to (H', t'_t)$ with $t_t = f_t\ u_t$ and $H = H_f \uplus H_u$. The reduction is a CTX rule of the form

$$\frac{(H_\square, t_h) \to^{\mathsf{hd}} (H'_\square, t'_h)}{(H_E \uplus H_\square, E[t_h]) \to (H_E \uplus H'_\square, E[t'_h])}$$

Either E is the empty context $\square$, which is the "head reduction" case, or E is non-empty, the "non-head" case.

Head reduction. The interesting case is when we have a head reduction in the target term: f_t is of the form $(\lambda x.\, g_t)$, and the reduction happens on the redex $(\lambda x.\, g_t)\ u_t$.

In this case u_t must be a value v_t, and f_t is also a value (a λ-abstraction), so by Fact 7 (Target value inversion) we know that:

- (H_u, u_t) is a value configuration (B_v, v_t) and u_s is a related value: $u_s = v_s \sim (B_v, v_t)$.
- (H_f, f_t) is a value configuration $(B_f, \lambda x.\, g_t)$.

More precisely, we know that f_s is of the form $L[\lambda x.\, g_s]$, and that the subderivation of $\lambda x.\, g_s \sim \lambda x.\, g_t$ within the derivation of $f_s \sim (H_f, f_s)$ uses a simple rule. This is necessarily the λ-abstraction rule, so we have $[\![g_s]\!] = g_t$.

Note that if we replace this subderivation $\lambda x.\, g_s \sim (\emptyset, \lambda x.\, [\![g_s]\!])$ by a derivation of $g_s \sim (\emptyset, [\![g_s]\!])$ (Fact 6), we obtain a derivation of $L[g_s] \sim (B_f, [\![g_s]\!])$.

Our initial hypotheses

$$t_s \sim (H, t_t) \to (H', t'_t)$$

became

$$L[\lambda x.\, g_s]\ v_s \sim (B_f \uplus B_v, (\lambda x.\, [\![g_s]\!])\ v_t) \to (B_f \uplus B_v, [\![g_s]\!][v_t/x])$$

We conclude this case of the proof with

$$L[\lambda x.\, g_s]\ v_s \to L[g_s[v_s/x]] \sim (B_f \uplus B_v, [\![g_s]\!][v_t/x])$$

where the last relation is obtained from $L[g_s] \sim (B_f, [\![g_s]\!])$ and $v_s \sim (B_v, v_t)$ by Lemma 12 (Relation substitution).

Non-head reduction. If $f_t\ u_t$ is not the main redex of the reduction, we want to proceed by induction on f_t or u_t, depending on where the reduction happens. Our evaluation context $E[\Box]$ must be of the form $E_f[\Box]\ u_t$ or $f_t\ E_u[\Box]$; the proof of both cases is symmetric, so we only discuss the case $E_f[\Box]\ u_t$.

$$\frac{f_s \sim (H_f, f_t) \qquad u_s \sim (H_u, u_t)}{f_s\ u_s \sim (H_f \uplus H_u, f_t\ u_t)} \qquad \frac{(H_\Box, t_h) \to^{\mathsf{hd}} (H'_\Box, t'_h)}{(H_E \uplus H_\Box, E_f[t_h]\ u_t) \to (H_E \uplus H'_\Box, E_f[t'_h] u_t\ )}$$

To proceed by induction on our premise $f_s \sim (H_f, f_t)$ we need the fact that $H_\Box$ is a sub-heap of H_f (rather than H_u). This is "obviously" the case, but we found it subtle to justify precisely.

For any variable $x \in \mathsf{dom}(H_\Box)$, let us show that x belongs to the domain of H_f and not H_u — the two heaps are disjoint. By Fact 3 (Relevant head reduction), we know that $H_\Box$ is included in the free variables of t_h, so in particular x is free in f_t. Remember that we assume that, in each syntactic object (term, configuration, derivation...), bound variables are distinct from free variables; this can always be chosen to be the case by performing α-renamings appropriately. In the derivation of $f_s\ u_s \sim (H_f \uplus H_u, f_t\ u_t)$, if x was not in $\mathsf{dom}(H_f)$ it would be free in (H_f, f_t); in particular, it could not be bound in (H_u, u_t), so we know $x \notin \mathsf{dom}(H_u)$.

We thus have that H_f is of the form $H_{E_f} \uplus H_\Box$, so we can use our induction hypothesis on

$$f_s \sim \frac{(H_\Box, t_h) \to^{\mathsf{hd}} (H'_\Box, t'_h)}{(H_f, E_f[t_h]) \to (H_{E_f} \uplus H'_\Box, E_f[t'_h])}$$

to get a f'_s such that

$$f_s \to^? f'_s \sim (H_{E_f} \uplus H'_\Box, E_f[t'_h])$$

which lets us conclude our goal with

$$f_s\ u_s \to^? f'_s\ u_s \sim (H_{E_f} \uplus H'_\Box \uplus H_u, E_f[t'_h]\ u_t)$$

Letrec rules.

Simulating the letrec rules is fairly simple, because the relation was precisely designed to correspond to the dynamic semantics of the translation of `letrec` in the target language, with four rules INIT, WRITE, DONE, FURTHER that correspond to four possible states of reduction of this translation, with reductions from one state only to itself or to the next state in the list.

Rule. INIT

$$\frac{J \neq \emptyset}{\begin{array}{c}\texttt{let rec}\ \left(x_i = t_i\right)^{i \in I}, \left(x_j = t_j\right)^{j \in J}\ \texttt{in}\ u \\ \sim \left([x_i \mapsto \bot]^{i \in I}, \texttt{new}\ \left(x_j\right)^{j \in J}\ \texttt{in}\ \texttt{par}(x_k \leftarrow t_k)^{k \in I \uplus J}; [\![u]\!]\right)\end{array}}$$

Any reduction of this target term is of the form below, with $J = 1 \uplus J'$ where 1 is some singleton set $\{\star\}$:

$$\left([x_i \mapsto \bot]^{i \in I}, \texttt{new}\ x_\star\ \texttt{in new}\ \left(x_j\right)^{j \in J'}\ \texttt{in}\ \texttt{par}(x_k \leftarrow t_k)^{k \in I \uplus 1 \uplus J'}; [\![u]\!]\right)$$

$$\rightarrow \left([x_i \mapsto \bot]^{i \in I \uplus 1}, \texttt{new}\ \left(x_j\right)^{j \in J'}\ \texttt{in}\ \texttt{par}(x_k \leftarrow t_k)^{k \in I \uplus 1 \uplus J'}; [\![u]\!]\right)$$

If $J' \neq \emptyset$, we have $t_s \sim (H', t'_t)$ again an instance of the INIT rule. If $J' = \emptyset$ then $\texttt{new}\ \left(x_j\right)^{j \in J'}\ \texttt{in}\ t$ is just t, so our reduced term is of the form

$$\left([x_i \mapsto \bot]^{i \in I \uplus 1}, \texttt{par}(x_k \leftarrow t_k)^{k \in I \uplus 1}; [\![u]\!]\right)$$

which is related to the same source term by the WRITE rule, with an empty set of committed values $\left(v_{s,i}, v_{t,i}\right)^{i \in I}$.

Rule. WRITE

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i \in I} \qquad \left(t_{s,j} \sim (H_j, t_{t,j})\right)^{j \in J}}{\begin{array}{c}\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i \in I}, \left(y_j = t_{s,j}\right)^{j \in J}\ \texttt{in}\ u \\ \sim \left([x_i \mapsto v_{t,i}]^{i \in I}[y_j \mapsto \bot]^{j \in J} \uplus \left(B_i\right)^{i \in I} \uplus \left(H_j\right)^{j \in J}, \texttt{par}((\texttt{Done})^{i \in I}, \left(y_j \leftarrow t_{t,j}\right)^{j \in J}); [\![u]\!]\right)\end{array}}$$

There are three possible kinds of reduction for a related target term of this form:

(1) We may be doing a reduction within one of the $\left(t_{t,j}\right)^{j \in J}$.
(2) If one of the $\left(t_{t,j}\right)^{j \in J}$ is a value $v_{t,j}$, the write $y_j \leftarrow v_{t,j}$ may be committed.
(3) If the set of uncommitted bindings $\left(y_j \leftarrow t_{t,j}\right)^{j \in J}$ is empty, the subterm $\texttt{par}(\texttt{Done})^{i \in I}$ reduces to Done.

In the first case, we have $(H_j, t_{t,j}) \rightarrow (H'_j, t'_{t,j})$.[9] By induction hypothesis we get $t'_{s,j} \sim (H'_j, t'_{t,j})$, and can conclude by using the WRITE rule again.

[9] It is not immediate that the part of the global heap that is modified is precisely H_j, but it the same reasoning that we detailed in the "Simple rules" case. We will omit discussing similar instances of this point in the rest of the proof.

In the second case, we have $J = 1 \uplus J'$ where $t_{t,\star}$ is a value $v_{t,\star}$, and thus $H_\star$ a value heap $B_\star$ by Fact 7 (Target value inversion). The reduction is thus of the form

$$\left(\begin{array}{l}[x_i \mapsto v_{t,i}]^{i\in I}[y_\star \mapsto \bot][y_j \mapsto \bot]^{j\in J'} \uplus (B_i)^{i\in I} \uplus B_\star \uplus \left(H_j\right)^{j\in 1\uplus J'},\\ \mathsf{par}((\mathsf{Done})^{i\in I}, \left(y_\star \leftarrow v_{t,\star}\right)^{\star\in 1}, \left(y_j \leftarrow t_{t,j}\right)^{j\in J'}); [\![u]\!]\end{array}\right)$$

$$\to \left(\begin{array}{l}[x_i \mapsto v_{t,i}]^{i\in I}[y_\star \mapsto v_{t,\star}][y_j \mapsto \bot]^{j\in J'} \uplus (B_i)^{i\in I} \uplus B_\star \uplus \left(H_j\right)^{j\in J'},\\ \mathsf{par}((\mathsf{Done})^{i\in I}, (\mathsf{Done})^{\star\in 1}, \left(y_j \leftarrow t_{t,j}\right)^{j\in J'}); [\![u]\!]\end{array}\right)$$

and we can use the WRITE rule again (with $I + 1$ and J') to relate to the same source term.

In the third case, the reduction is

$$([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I}, \mathsf{par}(\mathsf{Done})^{i\in I}; [\![u]\!]) \to ([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I}, \mathsf{Done}; [\![u]\!])$$

and the reduced target term is related to the initial source term by DONE.

Rule. DONE

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i\in I}}{\mathtt{let\ rec}\ \left(x_i = v_{s,i}\right)^{i\in I}\ \mathtt{in}\ u \sim ([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I} H, \mathsf{Done}; [\![u]\!])}$$

The only possible reduction of the target term is

$$([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I} H, \mathsf{Done}; [\![u]\!]) \to ([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I} H, [\![u]\!])$$

The resulting target term is related to the initial source term by the rule FURTHER, using Fact 6 () to relate u and $[\![u]\!]$.

Rule. FURTHER

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i\in I} \qquad u_s \sim (H, u_t)}{\mathtt{let\ rec}\ \left(x_i = v_{s,i}\right)^{i\in I}\ \mathtt{in}\ u_s \sim ([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I} \uplus H, u_t)}$$

The only possible reduction of the target term comes from a reduction $(H, u_t) \to (H', u_t')$. The proof in this case is by immediate induction hypothesis on $u_s \sim (H, u_t)$.

Heap rules.

Rule. HEAP-WEAKEN

$$\frac{t_s \sim (H, t_t) \qquad \forall x \in \mathsf{dom}(B),\ x \notin (H, t_t)}{t_s \sim (H \uplus B, t_t)}$$

By Fact 3 (Relevant head reduction), any head reduction $(H_\square, t_h) \to^{\mathsf{hd}} (H'_\square, t_h)$ has $H_\square$ included in the free variables of t_t, and $H'_\square$ in its free or bound variables. In particular, those two sub-heaps of $H \uplus B$ can be assumed disjoint from B, so they are sub-heaps of H, with $H = H_E \uplus H_\square$ and $H_E, H'_\square, B$ disjoint.

We can conclude by induction: from

$$t_s \sim (H_E \uplus H_\square, E[t_h]) \to (H_E \uplus H'_\square, E[t'_h])$$

we have by induction a t'_s such that

$$t_s \to^? t'_s \sim (H_E \uplus H'_\square, E[t'_h])$$

and we can conclude (as $H'_\square$ is disjoint from B) with the HEAP-WEAKEN rule again ($H_E, H'_\square, B$ are disjoint)

$$t_s \to^? \frac{t'_s \sim (H_E \uplus H'_\square, E[t'_h])}{t'_s \sim (H_E \uplus H'_\square \uplus B, E[t'_h])}$$

Rule. HEAP-COPY

$$\frac{t_s \sim (H, t_t) \qquad \phi \text{ compatible with } H \qquad \mathrm{dom}(\phi) \subseteq \mathrm{dom}(H)}{t_s \sim (\phi(H), \phi(t_t))}$$

This is the delicate case of proof, testing our definition of (ϕ compatible with H).

Our reduction must be a CTX rule, so $\phi(t_t)$ is of the form $E_0[t_{h,0}]$. Variable renamings preserve the term structure, so the preimage by ϕ of $E_0[t_{h,0}]$ must itself be of the form $E[t_h]$. Without loss of generality, we can thus assume that $\phi(t_t)$ is of the form $\phi(E)[\phi(t_h)]$.

We then reason by case analysis on the possible head reduction rules involving $\phi(t_h)$. In each case, we will refine this without-loss-of-generality reasoning by inverting ϕ on the specific structure of the head reduction rule.

Rule case: pure reductions. Without loss of generality, we can assume that any pure reduction from $\phi(H, t_t)$ must be of the form

$$\frac{(\emptyset, \phi(t_h)) \to^{\mathsf{hd}} (\emptyset, u'_h)}{(\phi(H), \phi(E)[\phi(t_h)]) \to (\phi(H), \phi(E)[u'_h])}$$

Substitutions preserve reductions: if $t \to t'$ in the λ-calculus then $t[\sigma] \to t'[\sigma]$ for any variable-to-terms substitution σ. In the case of variable renaming ϕ only, we have a converse property that if $\phi(t)$ is reducible, then t itself is reducible (a variable-to-variable substitution cannot introduce a redex). If $\phi(t)$ reduces to some u', then t reduces to some t'; as substitutions preserve reductions we furthermore have that $\phi(t') = u'$.

From our assumption $(\emptyset, \phi(t_h)) \to^{\mathsf{hd}} (\emptyset, u'_h)$ we thus have t'_h such that $\phi(t'_h) = u'_h$ and $(\emptyset, t_h) \to^{\mathsf{hd}} (\emptyset, t'_h)$. We thus have

$$t_s \sim (H, t_t) = \frac{(\emptyset, t_h) \to^{\mathsf{hd}} (\emptyset, t'_h)}{(H, E[t_h]) \to (H, E[t'_h])}$$

which gives by induction hypothesis a t'_s such that

$$t_s \to^? t'_s \sim (H, E[t'_h])$$

which lets us conclude by applying HEAP-COPY again:

$$t_s \to^? \frac{t'_s \sim (H, E[t'_h])}{t'_s \sim (\phi(H), \phi(E)[u'_h])}$$

Rule case: NEW

Without loss of generality, a NEW reduction of a configuration in the image of a variable renaming ϕ must start from a term of the form $\phi(H, E[\mathtt{new}\ x\ \mathtt{in}\ u])$. The bound variable x can be assumed outside the domain of H and the free variables of $E[\mathtt{new}\ x\ \mathtt{in}\ u]$, as well as outside their image through ϕ. We have $\mathrm{dom}(\phi) \subseteq \mathrm{dom}(H)$, so we know that x is also outside the domain of ϕ, which gives $\phi(x) = x$. Thus any such reduction can be assumed to be of the form

$$t_s \sim \frac{(\emptyset, \phi(\mathtt{new}\ x\ \mathtt{in}\ u)) \to^{\mathsf{hd}} ([x \mapsto \bot], \phi(u))}{(\phi(H), \phi(E)[\phi(\mathtt{new}\ x\ \mathtt{in}\ u)]) \to (\phi(H)[x \mapsto \bot], \phi(E[u]))}$$

From our premise

$$t_s \sim (H, E[\mathtt{new}\ x\ \mathtt{in}\ u])$$

we have by induction hypothesis a t'_s such that

$$t_s \to^? t'_s \sim (H[x \mapsto \bot], E[u])$$

and we can conclude with the HEAP-COPY rule again

$$t_s \to^? \frac{t'_s \sim (H[x \mapsto \bot], E[u]) \qquad \phi \text{ compatible with } H[x \mapsto \bot]}{t'_s \sim (\phi(H[x \mapsto \bot]), \phi(E[u]))}$$

where the compatibility of ϕ with $H[x \mapsto \bot]$ is a direct consequence of Fact 4 (Union compatibility).

Rule case: SET

Without loss of generality, a SET reduction from a term in the image of a variable renaming ϕ must be of the form

$$\phi(H[x \mapsto \bot], E[x \leftarrow v]) \to \phi(H[x \mapsto v], E[\mathsf{Done}])$$

We also have a premise $t_s \sim (H[x \mapsto \bot], E[x \leftarrow v])$ where the target reduces as

$$t_s \sim (H[x \mapsto \bot], E[x \leftarrow v]) \to (H[x \mapsto v], E[\mathsf{Done}])$$

which gives by induction hypothesis a t'_s such that

$$t_s \to^? t'_s \sim (H[x \mapsto v], E[\mathsf{Done}])$$

We can conclude by applying HEAP-COPY again on this last relation,

$$t_s \to^? \frac{t'_s \sim (H[x \mapsto v], E[\mathsf{Done}])}{t'_s \sim \phi(H[x \mapsto v], E[\mathsf{Done}])}$$

provided that ϕ is compatible with $H[x \mapsto v]$. It is (see Definition 7):

definedness is preserved from $H[x \mapsto \bot]$

functionality holds for any pair of references in H. For pairs containing x, we know by definedness of ϕ on $H[x \mapsto \bot]$ that $z \neq x$ implies $\phi(z) \neq \phi(x)$, which implies functionality.

REMARK 6. *This case is the reason why definedness was introduced in definition of compatible renamings. Functionality is required for the notation $\phi(H)$ to even make sense, but definedness is not an obvious requirement. Here is it essential for SET to preserve functionality.*

Rule case: LOOKUP

Without loss of generality, a LOOKUP reduction of a configuration in the image of a variable renaming ϕ must start from a term of the form

$$\phi(H[x \mapsto v], E[x]) \to \phi(H[x \mapsto v], E[v])$$

we also have a premise $t_s \sim (H[x \mapsto v], E[x])$ so we have the reduction

$$t_s \sim (H[x \mapsto v], E[x]) \to (H[x \mapsto v], E[v])$$

and thus by induction hypothesis a t'_s such that

$$t_s \to^? t'_s \sim (H[x \mapsto v], E[v])$$

which lets us conclude by applying HEAP-COPY again (we know by assumption that ϕ is compatible with $H[x \mapsto v]$)

$$t_s \to^? \frac{t'_s \sim (H[x \mapsto v], E[v])}{t'_s \sim \phi(H[x \mapsto v], E[v])}$$

□

C.3 Viciousness and Segfaults

Besides showing a tight correspondence between our local and global store semantics, the backward simulation result is intended to prove that this compilation strategy preserves safety: if our static analysis accepts a source term, then its compilation will not fail due to undefined values. To establish this, we relate the notions of failures in the target language (forcing contexts and Segfault terms from Figure 6) to failures in the source language (forcing contexts and Vicious from Figure 5).

Fact 8 (No value forcing).
A target value v_t is never of the form $E_{f,t}[t]$ for any target forcing context $E_{f,t}$.

Lemma 13 (Target forcing inversion).
If t_s is related to a variable-forcing target configuration

$$t_s \sim (H, E_{f,t}[x])$$

then

$$t_s = E_{f,s}[x]$$

for some source forcing context $E_{f,s}$.

Proof. A target forcing context $E_{f,s}$ is defined as the identity $\Box$ or the composition of an ordinary evaluation context E with a forcing frame F_f (see Figure 6).

The proof is by induction on the relation; for each layer of structure in the relation, it may be part of E, then we proceed by induction, of the forcing frame F_f, then it is a base case.

- variable case: immediate.
- abstraction case: impossible.
- application case: by direct induction if part of E, and immediate if part of F_f.
- constructor case: by direct induction.
- match case: as the application case.
- init: impossible as $\texttt{new}\ x\ \texttt{in}\ \Box$ cannot be the prefix of a forcing context.
- write: the target term is a forcing context if the context goes into one of the $\left(y_j \leftarrow t_{t,j}\right)^j$ being evaluated – $[\![u]\!]$ is not in reducible position. The corresponding source term $t_{s,j}$ is in reducible position in the source, so we proceed by induction.
- done: impossible.
- further: by direct induction.
- heap-weaken, heap-copy: by direct induction.

□

Lemma 14 (Definedness relation).
If

$$E_s[\Box] \sim_{\text{ctx}} (H, E_t[\Box]) \qquad\qquad (z = v) \in E_s$$

then $H(z)$ is defined and distinct from $\bot$.

Remark 7. *By $E_s[\Box] \sim_{\text{ctx}} (H, E_t[\Box])$ we mean the relation between contexts that extends $(\sim)$ with the rule $\Box \sim_{\text{ctx}} (\emptyset, \Box)$, or equivalently $E_s[x] \sim (H, E_t[x])$ for some variable x fresh for E_s, H, E_t.*

Proof. The proof is by induction on the derivation of $E_s[\Box] \sim_{\text{ctx}} (H, E_t[\Box])$. Most cases are immediate as they do not add bindings to the context.

- hole case $\Box$: immediate (there is no z such that $(z = v) \in \Box$).
- variable case: impossible (not of the form $E[\Box]$).
- application case: the holes must be on the same side in the source and target applications (as a hole is only related to a hole), so we can proceed by direct induction.
- construction case: similar to the application case.
- match case: by direct induction.
- init, done: impossible (the target is not of the form $E[\Box]$).
- write and further: these are the interesting cases, proved below.
- heap-weaken, heap-copy: by direct induction.

Rule. WRITE

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i\in I} \qquad \left(t_{s,j} \sim (H_j, t_{t,j})\right)^{j\in J}}{\begin{array}{c}\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i\in I}, \left(y_j = t_{s,j}\right)^{j\in J}\ \texttt{in}\ u \\ \sim \left([x_i \mapsto v_{t,i}]^{i\in I}[y_j \mapsto \bot]^{j\in J} \uplus (B_i)^{i\in I} \uplus \left(H_j\right)^{j\in J}, \mathsf{par}((\texttt{Done})^{i\in I}, \left(y_j \leftarrow t_{t,j}\right)^{j\in J}); [\![u]\!]\right)\end{array}}$$

In the target term, $[\![u]\!]$ is not in reducible position. The only related reducible subterms are the $\left(t_{t,j}\right)^{j\in J}$ in the target, which correspond to an evaluation context frame F of the form $\texttt{let rec}\ b, x = \Box, b'\ \texttt{in}\ u$ in the source. We never have $(x = v) \in F$ for this partly-evaluated frame, so z must come from the corresponding H_j and we proceed by induction on the corresponding premise $t_{s,j} \sim (H_j, t_{t,j})$.

Rule. FURTHER

$$\frac{\left(v_{s,i} \sim (B_i, v_{t,i})\right)^{i\in I} \qquad u_s \sim (H, u_t)}{\texttt{let rec}\ \left(x_i = v_{s,i}\right)^{i\in I}\ \texttt{in}\ u_s \sim \left([x_i \mapsto v_{t,i}]^{i\in I} \uplus (B_i)^{i\in I} \uplus H, u_t\right)}$$

If z is one of the $(x)_i^{i\in I}$, we have $[z \mapsto v_{t,i}]$ in the target heap so $H(z) \neq \bot$ as claimed. Otherwise z must be in a B_I or H, and we proceed by induction on the corresponding premise. □

Corollary 3 (Segfaults are Vicious).

$$t_s \sim (H, t_t) \quad \wedge \quad (H, t_t) \in \text{Segfault} \quad \Longrightarrow \quad t_s \in \text{Vicious}$$

Proof. (H, t_t) is in Segfault, so by definition it is of the form $(H'[x \mapsto \bot], E_{t,f}[x])$ for some target forcing context $E_{t,f}$ (Figure 6). In particular, we have $H(x) = \bot$.

By Lemma 13 (Target forcing inversion), the related source term t_s must be of the form $E_{s,f}[x]$ for some source forcing context $E_{s,f}$.

We cannot have $(x = v) \in E_{s,f}$, as by Lemma 14 (Definedness relation) this would imply $H(x) \neq \bot$. Therefore t_s is in Vicious (Figure 5). □

Theorem (5: Return-typed programs cannot segfault).

$$\emptyset \vdash t_s : \text{Return} \quad \wedge \quad (\emptyset, [\![t_s]\!]) \to^* (H, t_t') \quad \Longrightarrow \quad (H, t_t') \notin \text{Segfault}$$

Proof. The metatheory of our static analysis tells us that a Return-typed closed program t_s cannot reduce to a vicious term: if $t_s \to^* t_s'$, then $t_s' \notin$ Vicious.

We have $t_s \sim [\![\emptyset]\!] t_t$; by Theorem 4 (Backward Simulation), for any $(\emptyset, [\![t_s]\!]) \to^* t_t'$ there is a t_s' such that $t_s \to^* t_s'$ and $t_s' \sim (H, t_t')$. In particular, (H, t_t') cannot be in Segfault, as then the related t_s' would be in Vicious by Corollary 3 (Segfaults are Vicious). □